AN ANALYSIS OF THE EFFECTIVENESS OF A CONSTRUCTIVE INDUCTION-BASED VIRUS DETECTION PROTOTYPE

THESIS

Kevin T. Damp, Captain, USAF

AFIT/GCS/ENG/00J-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/00J-01

AN ANALYSIS OF THE EFFECTIVENESS OF A CONSTRUCTIVE INDUCTION-BASED VIRUS DETECTION PROTOTYPE

THESIS

Kevin T. Damp
Captain, USAF

AFIT/GCS/ENG/00J-01

AFIT/GCS/ENG/00J-01

AN ANALYSIS OF THE EFFECTIVENESS OF A CONSTRUCTIVE INDUCTION-BASED VIRUS DETECTION PROTOTYPE

THESIS

Presented to the Faculty of the Graduate School of Engineering and Management

of the Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Kevin T. Damp, B.S.

Captain, USAF

April 2000

AFIT/GCS/ENG/00J-01

AN ANALYSIS OF THE EFFECTIVENESS OF A CONSTRUCTIVE INDUCTION-BASED VIRUS DETECTION PROTOTYPE

Kevin T. Damp, B.S.

Captain, USAF

Approved:

[illegible]	12 APR 00
Dr. Gregg H. Gunsch	Date
[illegible]	12 APR 00
Dr. Gary B. Lamont	Date
John S. Crown	12 APR 00
Maj John S. Crown	Date

Acknowledgments

People who deserve thanks:

My thesis advisor, Dr. Gregg Gunsch.

My committee members, Maj John Crown and Dr. Gary Lamont.

Charlie Powers, Dave Doak, Mike Banford, Gary Mauersberger, Ron Adams, Mary Jane McCormick, the SC Plans Division and SC Customer Support section, and everyone else who supported me during my time as a student.

My fellow students.

My new in-laws, Chuck and Lynda LaFleur.

My parents.

And, of course, my wife, Charlene.

Table Of Contents

ACKNOWLEDGMENTS .... **I**

LIST OF TABLES .... **V**

LIST OF FIGURES .... **VI**

ABSTRACT .... **VII**

1. INTRODUCTION **1**

1.1. MOTIVATION 1

1.2. PROBLEM 3

1.3. RESEARCH OBJECTIVES 6

1.4. APPROACH 7

1.5. SCOPE 8

1.6. THESIS OVERVIEW 9

2. LITERATURE REVIEW **11**

2.1. OVERVIEW 11

2.2. COMPUTER VIRUSES 12

2.2.1. Characteristics and Structure *12*

2.2.2. Types *13*

2.3. ANTIVIRUS PROGRAMS 15

2.3.1. Measures of Effectiveness *17*

2.3.2. Types *18*

2.4. MACHINE LEARNING AND INDUCTION 20

2.4.1. Components of an Inductive Learning System *22*

2.4.2. Selective Induction *23*

2.4.3. Constructive Induction *24*

2.5. STATISTICAL ANALYSIS USING RESPONSE SURFACE METHODOLOGY 26

2.6. SUMMARY 27

3. SYSTEM DESIGN AND METHODOLOGY 28

3.1. OVERVIEW 28

3.2. GENERAL DESCRIPTION 29

3.3. OBJECTIVES 29

3.4. REQUIREMENTS 31

3.5. ARCHITECTURE 32

3.5.1. Subsystems *33*

3.5.1.1. Learning Engine 33

3.5.1.2. Scanner 34

3.5.1.3. Signature Database 35

3.6. DYNAMIC STRUCTURE 35

3.7. DATA FLOW 39

3.8. TESTING AND INTEGRATION 39

3.9. SUMMARY 40

4. SYSTEM IMPLEMENTATION 42

4.1. OVERVIEW 42

4.2. DEFINITIONS 42

4.3. INDUCTION ENGINE 44

4.3.1. Hypothesis Generation *44*

4.3.1.1. Hypothesis 45

4.3.1.2. Examples 45

4.3.1.3. Feature Extraction 46

4.3.1.4. Construction 48

4.3.2. Hypothesis Evaluation *51*

4.3.3. Hypothesis Ordering *53*

4.3.4. Hypothesis Incorporation *53*

4.4. SCANNING USING A DECISION TREE 54

4.5. SUMMARY 55

5. ANALYSIS AND RESULTS 56

5.1. OVERVIEW 56

5.2. TEST CASES 57

5.2.1. Verification Testing *58*

5.2.2. Laboratory and Operational Test Cases *60*

5.2.3. Future Test Cases *71*

5.3. PERFORMANCE 75

5.4. SUMMARY 76

6. CONCLUSIONS 77

6.1. OVERVIEW 77

6.2. RESEARCH HYPOTHESIS 78

6.3. IMPLICATIONS 80

6.4. LIMITATIONS 81

6.5. FUTURE RESEARCH 82

6.6. SUMMARY 83

APPENDIX A -- SOURCE CODE 85

BIBLIOGRAPHY 86

VITAS 88

List of Tables

TABLE 1 -- FEATURE EXTRACTION METHODS 47

TABLE 2 -- OPERATOR DEFINITIONS 49

TABLE 3 -- DATA FOR THE WEATHER EXAMPLE 58

TABLE 4 -- LABORATORY TEST CASE DEFINITIONS 60

TABLE 5 -- REVISED LABORATORY TEST CASE DEFINITIONS 61

TABLE 6 -- LABORATORY TEST CASE RESULTS 62

TABLE 7 -- OPERATIONAL TEST CASE DEFINITIONS 64

TABLE 8 -- OPERATIONAL TEST CASE RESULTS 65

TABLE 9 -- UNIQUE FEATURES IN OPERATIONAL TEST CASES 67

TABLE 10 -- PROPOSED TEST CASE A 71

TABLE 11 -- PROPOSED TEST CASE B 74

List of Figures

FIGURE 1 -- EQUIVALENT REPRESENTATIONS OF A CONCEPT 68

FIGURE 2 -- DECISION TREE FOR TEST CASE A USING SINGLE-OPERATOR HYPOTHESES 72

AFIT/GCS/ENG/00J-01

Abstract

Computer viruses remain a tangible threat to systems both within the Department of Defense and throughout the greater international data communications infrastructure on which the DoD increasingly depends. This threat is exacerbated continually, as new viruses are introduced at an alarming rate by the growing collection of connected machines and their operators. Unfortunately, current antivirus solutions are ill-equipped to address these issues in the long term. This thesis documents an investigation into the use of constructive induction, a form of machine learning, as a supplemental antivirus technique theoretically capable of detecting previously unknown viruses through generalized decision-making techniques. A group of examples derived from common software applications, utilities, and viruses was tested in order to evaluate the benefits of adding constructive induction to the process of selecting suitable virus signatures. A prototype virus detection system subcomponent, DRIVER, was developed to conduct the experiments. Due to the feature-rich content of nontrivial example files and DRIVER's ability to assemble decision trees, results showed marginal benefits--compounded with significantly increased computational resource requirements--in the use of constructive induction. Future research, emphasizing a combination of optimization techniques and test cases increasingly approximating "real world" detection scenarios, should eventually establish whether constructive induction represents a genuinely useful and practical alternative to today's antivirus measures.

AN ANALYSIS OF THE EFFECTIVENESS OF A CONSTRUCTIVE INDUCTION-BASED VIRUS DETECTION PROTOTYPE

1. Introduction

1.1. Motivation

Originally the topic of nothing more than academic discourse, computer viruses have slowly emerged as a persistent burden on computer operators across the globe, from the smallest businesses to the United States Air Force and Department of Defense [WKC95]. These viruses, much like their biological namesakes, are capable of reproducing and spreading without the knowledge or consent of computer operators—and, due to the computer industry's nonstop progress in the areas of interoperability and interconnectivity, are being provided with an increasing number of opportunities to do so [Kep94]. Another interesting and important parallel between digital and biological viruses is the possibility of a complete lack of symptoms on the part of the host organism or computer. Some viruses simply spread without altering the host in any other detectable fashion. Many computer viruses make themselves known to the operator in a wholly non-threatening or trivial manner [IUV99]. However, the possibility of a virus introducing destructive or otherwise malicious code—whether intended or not—is, of course, more than sufficient motivation for finding and eliminating all viruses of any type.

In order to intercept and disgorge viruses on behalf of the user, a number of antivirus programs have been developed, most of them for microcomputers, since the overwhelming majority of known viruses affect only those platforms. These programs currently are limited to the following techniques: [IUV99]

- *Scanning* – The antivirus program simply checks files and boot records (the two data types that viruses modify) for the byte patterns of known viruses. This feature must be carefully implemented, since the indicator byte patterns cannot appear in normal program files.
- *Behavior Blocking* – The antivirus program detects changes to a boot record or file and watches the system for distinctive behavior common to certain classes of known viruses (such as writing to the hard disk at an unusual time), alerting the user if suspicious activity has occurred. Since these actions are often legitimate system functions, the user must know when the reported change is indicative of a virus. This form of protection is sometimes referred to as *change detection* or *heuristic analysis.*
- *Integrity Monitoring* – The antivirus program uses a database of file information (size, date last modified, and so forth) to periodically check the system for

anomalies that may be indicative of a virus. In order for this technique to be effective, the system must be "clean" when the database is established.

- *Verification* – Separate code within the antivirus program acts as a follow-up to one of the above techniques, making a final determination as to the identification and location of the virus. This can only be accomplished for viruses that have been carefully analyzed.
- *Disinfection* – The program may be able to remove a virus that has been identified and located, repairing the affected files. This is a delicate process, which requires specific knowledge about the nature of the particular virus. If a virus has been incorrectly identified, the process of attempting system repair is likely to cause further damage. This part of the antivirus program must be extremely reliable.

Chapter Two of this document presents some of the information regarding viruses and antivirus programs introduced above in greater detail.

1.2. Problem

Although antivirus programs in their present form have been of great benefit to the user community, evidence suggests that new techniques will soon be required in order to combat the growing virus population. Human virus analysts have become heavily overburdened with the task of evaluating new viruses and devising methods to combat them. This trend represents a potential increase in the amount of time required for an end user to acquire defensive measures against new viruses. Unfortunately, the amount of time required for an infection to become widespread continues to *decrease,* as microcomputers become increasingly connected and interoperable. These factors, and the many limitations of current antivirus products have prompted researchers to explore new techniques for identifying and removing unwanted software, including both viruses and other forms of code such as Trojan Horses, which can cause equally serious damage, but cannot technically be classified as viruses unless they are capable of automatically reproducing themselves. [Kep94]

An emerging paradigm for handling undesirable software is the use of models based on the immune systems of living organisms (and vertebrates in particular) [FHS97]. Such systems are capable of handling a variety of intruders without the need for input from external agencies. IBM researcher Jeffrey Kephart summarized the advantageous properties of biological immune systems: [Kep94]

- Recognition of known intruders
- Elimination/neutralization of intruders
- Ability to learn about previously unknown intruders
    - Determine that the intruder doesn't belong
    - Figure out how to recognize it
    - Remember how to recognize it
- Use of selective proliferation and self-replication for quick recognition and response

Certainly, one of the most remarkable attributes identified was the ability to learn about intruders not previously encountered by the system. For a computer to emulate this characteristic, some form of machine learning must take place without user intervention. One such technique is that of selective induction, in which the machine attempts to classify test subjects based on properties exhibited by those subjects [BlM99]. For example, a computer attempting to find all the red fire trucks in a box of children's toys might first classify the toys by weight, then by the presence of wheels, and finally by color. Having successfully separated the red fire trucks from the other toys (probably after any number of failures) by applying a set of chosen characteristics, the computer will then "remember" how to achieve this classification. A problem arises when selective induction fails to successfully classify a group of examples, which can happen any time a group of available descriptors is insufficient to achieve the desired discrimination. These cases give rise to a related but more advanced technique known as *constructive induction*, in which operators are chosen and applied to the available attributes in order to construct new attributes. For example, while the properties of "pressure" and "volume" would be insufficient to identify any particular instance of an ideal gas, these properties can be combined, using the multiplication operator, to form a new attribute. Since P x V equals a constant for samples of an ideal gas, this technique provides an accurate and concise method of classifying ideal vs. non-ideal samples [Gun91]. When this technique is combined with proper knowledge of the problem domain, hypotheses can be formed and evaluated with the intent of forming an increasingly focused concept description. The result is a promising form of machine learning that may be capable of emulating some of the best features of biological immune systems. Both selective and constructive induction will be discussed in greater detail in Chapter Two.

Unfortunately, there are other problems inherent to the application of constructive induction. It is a computationally complex process in which billions of hypotheses can be formed through only a few construction iterations, placing a tremendous strain on the resources of the host computer. Both disciplined application of domain knowledge and carefully designed software are needed to enable practical virus detection using this learning method. A *prototype* software application called DRIVER, designed to function as a learning component in an integrated virus detection solution, was developed in conjunction with this thesis as a means of exploring some of the issues associated with the use of constructive induction in the antivirus problem domain. The proposed solution uses scanning as a means of detection, and operates only at the *local* level (i.e., on one machine at a time).

1.3. Research Objectives

This research represents a combination of the computer virus detection domain with that of inductive learning. Previous work of a similar nature--in particular, the research completed by AFIT graduates Cardinale and O'Donnell in 1999--serves as a baseline [CaO99]. These two researchers proposed both that the public health system was a useful model for a broad-reaching "Computer Health System" and that the human immune system was a suitable model for the virus detection system of an individual computer. They further theorized that constructive induction is a suitable basis for the virus detection system, holding promise as a method of automating the discovery of new viruses. This thesis represents an attempt to determine the extent to which these intuitions were correct. The nature of this research therefore implies the following hypothesis:

> Constructive induction is an appropriate foundation for the virus detection subsystem of a single-computer antivirus solution.

The primary research objective was to test this hypothesis in order to arrive at conclusions as to its validity. The next section provides a brief overview as to how this was accomplished. Chapter Three of this document explains this process in detail.

1.4. Approach

This research was conducted in two primary phases. The first of these was the research, development, and interim testing of the prototype constructive induction-based learning engine, DRIVER. This partially-implemented software application represents a component in a larger overall virus detection model, which also includes a file scanner and a database that serves as a repository for information about viruses. An evaluation of the requirements and objectives for DRIVER was accomplished first, followed by a high-level design of the modules, interfaces, and architecture of the software. Initial test cases and criteria were also identified during this stage. Implementation of the prototype--

using the Java programming language--took place in an iterative fashion, with individual routines being tested before integration within larger control structures. Each new version of DRIVER incorporated fixes for known problems, performance enhancements, and additional functionality.

The second research phase consisted of final testing, experimentation, and an analysis of the results. Following acceptance testing for functionality and correctness, experiments were performed to test the research hypothesis. In these experiments, DRIVER evaluated a series of test groups developed by Cardinale and O'Donnell. The test groups were divided into two categories: laboratory cases consisting of synthetic data intended to further test the functionality and correctness of the learning engine, and operational cases consisting of file segments culled from actual applications, utilities, and viruses. DRIVER utilized both selective and constructive induction in an attempt to classify the file segments within each test group. The largest share of attention was given to the operational test cases and their results, since these cases best represented scenarios likely to be encountered in an operational computing environment. Both the test cases and an analysis of the results are presented in Chapter Five and Chapter Six.

1.5. Scope

This research does not represent an effort to tackle the overall problem of providing timely, comprehensive protection from viruses and other forms of potentially malicious code to computer operators worldwide. It will instead apply a chosen machine learning technique—constructive induction—to further explore the potential for alternative virus detection and removal methods at the *local* level. More specifically, focus will be placed on a series of test cases indicative of the types of files likely to be encountered by a fully implemented antivirus solution operating in a typical computing environment. However, the techniques explored here might be applicable not only to virus detection, but also to the problem of automating the process of distinguishing between "self" and "nonself" on a particular machine, and possibly even other research areas within the domain of computer and network security. Chapter Six presents some suggestions as to how this line of research may be extended in several productive directions.

1.6. Thesis Overview

This document presents a detailed discussion pertaining to the research effort outlined in this chapter. Chapter Two is a review of some of the available literature regarding those domains applicable to this research; that is, viruses, methods of virus detection, the process of inductive learning as conducted by machines, and statistical analysis. In Chapter Three, the methodology used to design and build the constructive induction-based learning engine component is presented, followed by an in-depth look at its implementation in Chapter Four. The testing and analysis results are delivered in

Chapter Five. Finally, Chapter Six conveys the conclusion that, as implemented in the DRIVER software, constructive induction does *not* appear to be an efficient method of augmenting the virus detection process. As noted previously, this final chapter also provides suggestions as to avenues of future research.

2. Literature Review

2.1. Overview

This chapter presents the most important areas of study from which pieces of knowledge were needed in order to develop the constructive induction-based learning engine prototype. The chapter is divided into four principal sections. The first section is an introduction to computer viruses—how they work, how they are structured, and the different types of viruses known to researchers today. The second section concentrates on virus detection methods and the uncertain future faced by those methods most commonly in use at this time. The third section describes inductive learning, both in general and in how it can be applied to machines, with emphasis on the core process in the learning engine prototype, constructive induction. The fourth section is intended to briefly familiarize the reader with a statistical process optimization tool known as Response Surface Methodology (RSM). Although not a component of the learning engine prototype, RSM was used by previous researchers in an informal attempt to help guide portions of the constructive induction process, thereby reducing its computational complexity.

2.2. Computer Viruses

In order to build a virus detection system, one must first have a good understanding as to the nature of viruses [Lud98]. The domain of existing viruses continues to diversify over time as the total number of viruses introduced in the laboratory and/or "in the wild" continually increases. The following sections describe some of the characteristics that all viruses have in common, the structure typically exhibited by a virus, and the ways in which viruses can be classified into various families.

2.2.1. Characteristics and Structure

There are two characteristics exhibited by all forms of viruses--they are composed of executable code and have the ability to reproduce [NCS96]. "Executable," in this case, refers not only to hardware-native binary instructions, but also to any code, such as macros and scripts, capable of automatically triggering an event. Reproduction is enabled once the virus has attached itself to a machine or another program, since this provides access to the resources necessary for accomplishing the replication. Note that although infection continues automatically once the viral code is granted access to system resources, the original infection requires user intervention [Che97].

Typically, reproduction is not the only function accomplished by viral code. Many viruses also carry a *payload*, or a second executable block designed to do something other than ensure replication. The payload is often executed once the virus has successfully attached itself to another file or program. The range of functions that can be accomplished by the payload is equivalent to those possible using "ordinary" software applications, and is limited only by the skill and restraint of the virus author. Some payloads do nothing more than display a message to the user of the infected machine, notifying them of the virus' presence. Others intentionally erase or modify files on the host computer's hard disk. Still others may actually perform functions beneficial to the host machine, but because they do so in a clandestine fashion and without the consent of the user, such "help" is rarely appreciated. [WCC89]

Successfully infecting new files and programs requires that the virus contain routines for both locating new targets and, once found, reproducing itself within the target [Lud98]. Although these routines may be quite sophisticated, virus authors often limit their functionality, since a more simple program takes up less space within the infected file or program, and is therefore more difficult for antivirus software to detect.

2.2.2. Types

There are a number of ways in which viruses can be classified. Here are a few of them:

- By the operating system which the virus was designed to run under

- By the type of data or program which the virus tends to infect
- By the amount of damage the virus tends to cause
- By the methods of infection and/or reproduction
- By the method(s) used to avoid detection

Yet another group of classification categories are the environments in which the various viruses reside. These categories include network viruses, macro viruses, boot sector viruses, and file infector viruses. In years past, the most common of these was the boot sector virus, which, as the name implies, either resides in the boot sector of a magnetic storage disc, or changes the pointer to the boot sector [Kas99]. In recent years, macro viruses have grown substantially in prominence. They are designed to infect the data files associated with certain software applications that allow *macros*, or "scripts" which automatically execute a series of commands [AOG00]. Since data files are more commonly exchanged among users than executable files, files containing macro viruses are often provided ample opportunity to infect other systems. Network viruses are those that spread using e-mail, transport protocols, or some other facet of data communications [Kas99]. This is another fast-growing category, owing particularly to recent, dramatic increases in the number of machines connected to the Internet. Finally, file infector viruses are those which attach themselves to executable files, create duplicate files, or take advantage of features present in the file system used by a particular machine in order to reproduce. This is the type of virus that DRIVER--the learning engine prototype implemented in order to conduct this research--was designed to detect. File infectors can be further divided according to infection method. Those that use *overwriting* simply destroy the target file, replacing it with the viral code. *Parasitic viruses* use a somewhat more subtle approach--instead of rendering the target file unusable, the virus attaches itself to the beginning, end, or somewhere in the middle of the target. *Companion viruses* make no modifications to the target. The target is copied, and the viral code moves into the original spot in the file system. *Link viruses* operate in a similar fashion, but instead of moving the original file, they simply alter the file system's allocation table so that any attempt to run the target program actually results in virus execution. [Kas99]

Other types of viruses exist which cannot be so easily categorized by researchers. Some of these use sophisticated methods of masking their presence from detection software or other threats. For example, *polymorphic viruses* attempt to survive by executing encryption algorithms that continually change the binary signature of the virus itself [AOG00]. In other words, a polymorphic virus retains no identifying signature. Since the majority of antivirus applications, including the three-component system described in the following two chapters, use byte scanning to detect the presence of viruses, polymorphic viruses can be difficult to identify.

2.3. Antivirus Programs

Antivirus programs are the only means of defense most users have against the increasingly large and diverse collection of viruses that exists today. Thanks to the continuing efforts of vigilant users, analysts, and antivirus software developers, these programs provide a great deal of protection. However, the perfect antivirus solution--one capable of detecting every possible virus--does not exist. In fact, Cohen was able to prove that the existence of an algorithm that distinguishes between viruses and non-viruses without error is impossible [KCW83]. However, the pursuit of better antivirus techniques is not without merit. The following sections examine both the methods currently used to combat viral activity and the issues facing the antivirus industry.

At present, virus detection begins in the laboratory, where an analyst dissects a piece of assembly code known to represent a previously unexamined virus, or, less frequently, a new class of viruses [Kep94]. Once this expert has precise knowledge of the workings of the virus, including the method by which it spreads, the process of building a defense begins. In order to provide the greatest amount of protection to end users, two algorithms should be developed--one to detect the virus and one to repair any damage it has caused to an infected computer. In some cases, there is no chance to repair the damage caused by the virus. In any event, the new defense methods must then be delivered to end users, who, in turn, must update their antivirus software so as to incorporate the new data. Compared to the amount of time in which a virus can damage valuable systems, possibly causing data loss, the amount of time required to complete the defense cycle outlined above is understandably frustrating.

The learning engine developed in conjunction with this research was not designed to address the damage repair problem, only the detection problem. As discussed above, one of the foremost issues regarding current antivirus techniques is that they are reactive--a defense must be developed *after* the new virus has been discovered. Constructive induction, the process used by DRIVER to classify a group of example files, represents a solution that *may* reduce the need to rely upon third-party analysts and manual product updates in order to detect previously unseen viruses. For more information on this process, see the section on inductive learning in this chapter.

2.3.1. Measures of Effectiveness

Three ways of determining the effectiveness of an antivirus system are to measure its quality of detection, reliability, and completion speed [EAV99]. Detection quality encompasses all of the ways in which the antivirus software can correctly and incorrectly classify the files on a given system. Given a laboratory setting in which the true number of viruses and clean files on a test system are known in advance, there are three different ways in which detection quality can be measured. The first, most obvious of these is detection rate, or the number of viruses found as compared to the total number of viruses on the system. The second measure of detection quality is the false positive rate, or the number of files incorrectly classified as containing viral code over a certain number of total files. The third is the false negative rate, or the number of virus files missed by the scanner over the number of total files.

Reliability, the second measure of effectiveness, represents the ability of the antivirus application to scan *all* designated files without halting unexpectedly or requiring additional inputs from the user. A less than perfect reliability score means that there were still files remaining on the system which had not yet been scanned.

The final measure of effectiveness is simply the speed at which the antivirus program checks the computer. Since most users need their computers for other tasks, there are practical limits as to how long a complete system scan should take.

2.3.2. Types

The three detection methods used by virtually all modern antivirus applications are scanning, integrity monitoring, and behavior blocking. In practice, these methods are often combined within a single program for maximum versatility [AAV99]. Each of the three methods is discussed below.

Scanning is the process of exhaustively checking the contents of the memory and storage devices on a system to look for known viral code [AAV99]. The scanner uses a database of small byte sequences known as *signatures* to search for infected files. Each signature is known to appear in a certain virus or family of viruses. As previously discussed, the signatures can only be obtained through the effort of one or more expert human analysts, and once new signatures are added to the scanner's database, end users must somehow replicate the new database (update the antivirus software) locally. The input files used by DRIVER to classify groups of examples are the result of applying scanning techniques to existing chunks of code. Some scanners operate only upon a request from the user; others remain in memory and scan new or modified files automatically, as they are accessed. Remaining resident in memory is usually preferable, since the program can provide instant feedback regarding the contents of new files.

Another detection method used by some applications is *integrity monitoring.* In order to use this technique, an initial database of file size, checksum, and modification date information must be established and maintained as needed. Following that, the monitor periodically checks the files on the system and generates an alert if any illegitimate changes to a file are discovered [AAV99]. One of the primary drawbacks of integrity monitoring (used alone) is that viruses cannot be detected in a timely fashion--only when the infected file happens to be scanned. This gives the virus time to spread or conduct similarly undesirable activities.

Behavior blocking is the last detection method that will be discussed here. This type of program is always resident in system memory, watching for possible signs of virus activity on a machine or network [AAV99]. For example, a system call to overwrite a portion of the contents of an executable file may signal the presence of a file infector virus. One of the advantages of behavior blocking is that this type of activity can be detected immediately. However, sometimes this type of behavior represents legitimate system activity, and in general, behavior blocking detectors may generate false alarms for a wide range of system activities. For that reason and others, this detection method remains less popular than the two methods previously discussed.

2.4. Machine Learning and Induction

Machine learning is a branch of artificial intelligence concerned with the study of automated problem solving performance improvement [ABKS94]. This field of study can be divided into the five primary paradigms by which machine learning is historically accomplished, which are analytic learning, instance-based learning, genetic algorithms, neural networks, and inductive learning [LaS95]. Analytic learning is the process of synthesizing large amounts of knowledge in order to derive both immediate solutions and rules which can be applied to similar problems. Instance-based learning is a related technique in which ways of relating sets of specific cases and experiences to general problem forms are found. Genetic algorithms form rules by combining or transforming individual objects within one or more existing rule sets, a process which models the biological phenomenon of chromosome mutation. Neural networks represent an attempt to model the human brain using a multitiered network of nodes; knowledge is represented as a "network of units that spreads activation from input nodes through internal units to output nodes" [LaS95]. Inductive learning is a systematic method of discovering the characteristics and rules best capable of describing a problem, using groups of examples which have been labeled as either positive or negative. This is the general process that DRIVER uses to learn how to distinguish between "self" and "nonself" example files. For that reason, this chapter focuses additional attention on inductive learning and its variants.

Induction begins with a *concept*, which is a way of dividing a group of objects into categories. Useful concepts are often intuitive, which, unfortunately, does not make them any easier for a machine to understand. The process of induction is an attempt to "learn" the concept by observing a group of examples and determining which example attributes can be used to achieve the intended classification. Any given example that is a member of the group described by the concept is labeled as *positive*; otherwise, the example is *negative*. The optimal result of induction is knowledge sufficient to correctly classify not only all of the examples which have already been observed, but also any examples which may be introduced at a later time. The capability to distinguish between positive and negative examples is known as *discrimination* [Gun91]. However, in practice this result is often difficult to achieve.

Given a set of positive and negative examples, each exhibiting certain attributes, the process of inductive learning will find those attributes best capable of distinguishing between those examples. Upon completion of this learning phase, the machine is left with at least one *hypothesis*, which is a description approximating the concept with respect to the values of the example attributes. For example, if the concept is "motorcycles" and the example set is "vehicles," the machine may develop a hypothesis that motorcycles can be distinguished from other vehicles because they have two wheels. As the name implies, hypotheses are not guaranteed to be correct, and can be changed if proven wrong by subsequent examples. The hypotheses resulting from induction are collectively known as a *rule base* [CaO99].

The two types of inductive learning are selective induction and constructive induction. Selective induction uses only attribute values that have been *selected*, or acquired directly from the set of examples [BlM99]. Constructive induction uses not only the selected attributes, but also *constructed* attributes, which are derived from the selected attributes according to a specified formula. The addition of constructed attributes can be helpful when the original attributes are found insufficient to classify a particular group of examples. The following sections elaborate on these two forms of induction and the components found in an induction system.

2.4.1. Components of an Inductive Learning System

The components that all inductive learning systems have in common are a problem representation, a set of examples, construction rules, and "built-in" knowledge known as *bias* [Pro92]. The examples are described using the chosen problem representation. One of the key pieces of the problem representation is the set of example attributes. Most objects have a large number of measurable attributes, but only some of these are likely to be useful in the chosen problem domain. Since attribute selection can have a dramatic impact on the performance of the induction system, a human analyst with experience in the problem domain often decides which attributes will be used as components in the problem representation. Another issue regarding representation is structure--that is, how the attributes are arranged and maintained within the system [DiM81]. No particular structure works best in every case. In those systems that make use of constructive induction, construction rules constrain the ways in which the selected attributes may be utilized in building new attributes. The example set should be chosen carefully, since the system will use this data to determine how to distinguish between future examples. The examples are normally divided into two groups--the training set and the testing set. Induction is first performed using the training set, and then the testing set is used to evaluate the results.

2.4.2. Selective Induction

Given a collection of example attributes, selective induction seeks to discover a subset of one or more attributes best capable of describing the chosen concept. Because the attributes can be applied in combination with one another, the process is capable of making implicit use of conjunction, disjunction, and negation in relating the attributes to one another. Such a representation is possible *only* through the use of a data structure, such as the decision tree, which can be stored in either conjunctive or disjunctive normal form and interpreted accordingly. The relationships are defined as follows: two attributes joined using conjunction are interpreted as a single proposition using the logical operator *and*, so in order for the entire expression to be true, both of the named attribute values must be properties of the example. Attributes combined using disjunction are joined using logical *or*--at least one of the proposed attribute values must hold. Negation simply means that the concept does not include a particular attribute value. DRIVER uses a binary decision tree data structure to represent these relationships. Selective induction is limited to the use of these combinatorial techniques only. This is what separates selective induction from constructive induction--the latter is not limited in its ability to designate relationships between attributes [Ren90].

2.4.3. Constructive Induction

Although constructive induction is similar to selective induction, there is an important distinction between the two processes: constructive induction is capable of transforming the original problem representation by creating associations between the regions inherent to that representation [Ren90]. The mechanism by which these associations are established is the *operator*.

The hypothesis is one of the central data components of inductive learning. One way to describe the steps necessary for accomplishing constructive induction is to break the process down into the four ways in which hypotheses are manipulated: hypothesis generation, hypothesis ordering, hypothesis evaluation, and hypothesis incorporation. The hypothesis generation step is the point at which new hypotheses are created. Generation is not arbitrary; only existing attributes are used, and predefined construction methods dictate the types of relationships that will be established among attributes. The available operators represent one type of bias built into this process, and knowledge of the problem domain can be used to introduce other useful forms of bias. For example, if a particular operator is known to be of little use in a given problem domain, construction using that operator can be skipped during the generation phase, saving time and resources. The hypothesis ordering step is an attempt to constrain the total number of hypotheses that will be evaluated, since the generation step often produces an enormous number of hypotheses. This preliminary check is not thorough, but it is necessary from a practical standpoint, since a thorough examination of all the hypotheses would introduce another computational expense. Rather, a heuristic approach is used, which does not guarantee that useful hypotheses will not be inadvertently discarded [Gun91]. Therefore, the ordering algorithm represents another form of bias within the system. The hypothesis evaluation step takes the remaining group of hypotheses and tests each of them to discover which are most applicable to the stated concept. The method by which this evaluation takes place varies, but no matter what measure of effectiveness is used, every member of the example set must be examined against every hypothesis. That is, given *n* examples and *m* hypotheses, the running time of the evaluation step is $O(mn)$ [CLR90]. This is why the number of example and hypothesis inputs to the evaluation step should be constrained whenever possible. Finally, once the evaluation process has selected the most capable hypotheses, these selections are incorporated into the rule base. This is referred to as the hypothesis incorporation step. Even within the rule base, the individual hypotheses may be stored in order according to the results of the evaluation step.

Although the steps of the constructive induction process were presented in a certain order here, not all applications of the process accomplish the steps in this same order. Consider, for example, a constructive induction implementation in which only a fraction of the total number of hypotheses is generated at a time. In such a system, the smaller group of hypotheses might be evaluated first, and then ordered according to the evaluation scores. The incorporation step, then, might involve placing only a certain percentage of the top-scoring hypotheses in the rule base. In a subsequent iteration of this downscaled process, the incorporation step might be skipped if none of the newly-generated hypotheses are equivalent in power to the hypotheses already residing within the rule base. In its current implementation, DRIVER bypasses the hypothesis ordering step in favor of a complete evaluation of all available hypotheses. However, the user can choose to limit the complexity of the constructed hypotheses in order to reduce the amount of time needed to perform the evaluation step.

2.5. Statistical Analysis using Response Surface Methodology

Response surface methodology (RSM) refers collectively to a group of statistical techniques useful for accomplishing empirical model building and exploration [Mon91]. This process uses designed experiments in an attempt to characterize the relationship between one or more *predictors*, or input variables, and a *response.* The objective is to optimize the response variable. For example, RSM could be used to study the effects of pressure and temperature on a chemical reaction in which the desired outcome is to maximize the yield of the substance produced by the reaction. RSM can be of great benefit in analyzing phenomena that are not currently understood to the degree that permits models derived directly from physical mechanisms. Note, however, that the usefulness of the results depends directly on the *quality* of the experiment from which the data is derived, and the *accuracy* with which the conditions affecting the outcome of the experiment are measured [BoD87].

2.6. Summary

Information from several different areas of study must be coalesced in order to explore the hypothesis of this research. This chapter presented several of the most relevant associated topics. The types and functions of currently known computer viruses were explained, along with the most common ways in which they are detected and removed. Emphasis was placed on the need for new, more proactive methods of finding viruses and other potentially damaging types of undesired code. Forms of machine learning, especially the related processes of selective induction and constructive induction, were introduced and described in order to lay the foundation for the core component of a proposed virus detection subsystem, the design and implementation of which will be the central topic of the next two chapters. In Chapter Five, the structure and results of the experiments used to test the research hypothesis will be covered in detail, along with an analysis of those results. Chapter Six presents the conclusions that can be drawn from these experiments, and elaborates briefly on the most logical avenues of exploration for researchers intending to follow up this study.

3. System Design and Methodology

3.1. Overview

Chapter Two provided a brief examination of the topics relevant to this research, including constructive induction, the known forms of machine learning, and the current state of the art in virus detection and removal. This chapter synthesizes these concepts to describe the essential components of a localized (single processor) virus detection system utilizing techniques associated with machine learning. A discussion of the requirements, goals, and high-level architecture associated with the proposed system provides context. Note that this discussion revolves around a *fully operational* virus detection system. The DRIVER prototype, as actually implemented, does not perform all of the functions described in the following pages. Differences between DRIVER and the virus detection model as noted as appropriate. Chapter Four presents the components of this system in greater elaboration.

3.2. General Description

The DRIVER software represents an attempt to perform some of the functions of "traditional" antivirus programs using nontraditional methods. Antivirus software is typically made up of two components: a virus scanning engine and a database. The database consists of either explicit virus descriptors known as *signatures*, rules describing various forms of behavior associated with viruses, or some combination of the two. The DRIVER prototype is a learning engine capable of using hypotheses--selected or constructed from byte strings contained in files or file segments--in the construction of a *decision tree*, which provides a basis for the classification of files encountered at a later time. In the proposed virus detection system containing DRIVER, decision trees would be the tools residing in the signature database, and the scanner component would make use of them in determining the virus content of any scrutinized file.

3.3. Objectives

The definition of a system begins with its objectives. In the most general of terms, the goal of the DRIVER software development and testing process is to explore the application of a machine learning technique to the problem of detecting files—or even pieces of files—which do not belong on a particular computer. This domain includes viruses, Trojan horses, sniffers, and many other types of software employing a variety of techniques, but sharing an important trait—they all accomplish a function neither intended nor desired by the computer's operator (note, however, that I will often use the term "virus" as a generalization). Although current commercial antivirus products attempt to address this requirement, they all share some less-than-desirable characteristics, including a lack of proactive countermeasures and the need for manual updates to the signature database. In order to transcend these limitations, DRIVER was conceived with the following objectives in mind:

- *Learn* – Through an initial examination of examples, acquire information about the types of files which are and are not appropriate on a given computer system.
- *Detect* – Once trained, be capable of identifying unwanted code, including files not encountered in the past. This capability is a consequence of the inductive generalization that takes place during the learning stage.
- *Adapt* – Adjust to evolving system configuration data (i.e., *re*learn) and capitalize on past experience in order to identify suspicious files.
- *Remember* – Quickly recognize previously detected viruses.

The adaptation objective exists solely to illustrate the proposed detection system's ability to automatically relearn the concepts of self and nonself, a capability which is very limited in current antivirus products.

3.4. Requirements

Using the learning objective as a starting point, a group of more specific requirements for the proposed detection system can be developed and used as a basis for testing [BlF90]. They are:

- Learn about existing "self" (desired) and "nonself" (undesired) files through the examination of examples. Obviously, this requirement is directly derived from the first goal above.
- Identify useful *hypotheses*, if any, from provided examples using the selective induction process.
- Systematically search for more powerful hypotheses using the constructive induction process.
- Mitigate the computational complexity associated with constructive induction by utilizing domain knowledge and guidance obtained through the use of analysis techniques such as response surface methodology. (The existing learning engine implementation does not capitalize on this technique--see Chapter Six for suggestions regarding future research.)
- Drawing from a pool of available hypotheses, establish a binary decision tree capable of automatically classifying new examples, while restricting its total size to conserve machine resources.
- Add the decision tree to a signature database, which shares data with the scanner component (This feature has not been implemented in DRIVER; see next section for an explanation of the detection components).

Note that for research purposes, these functional requirements help form a baseline for the software that will be used to evaluate the usefulness of constructive induction across a series of test groups containing self and nonself examples. In previous research, effectiveness scores for constructed hypotheses, combined with an application of response surface methodology (RSM), provided some indication that the XOR and OR operators provided the greatest amount of classification power over this particular group of examples. The exploration of RSM represented one possible way to curb the explosion in the total number of hypotheses generated as a result of constructive induction. By showing trends in the classification power associated with batches of constructed hypotheses, RSM *may* provide performance benefits to the learning component of a detection system, allowing a reduction in the time needed to execute the generate-order-evaluate-incorporate process for large batches of hypotheses in many cases [CaO99]. Therefore, the DRIVER prototype was written, in part, to help substantiate this.

3.5. Architecture

A system's architecture has been described as, "The highest-level concept of a system in its environment" [Mar97]. The architecture of the detection prototype follows directly from those objectives and requirements stated above. Its general design stems from the need to collect, generate, and process large amounts of system and virus data. Since DRIVER may conceivably act as but one participant in a broader security scheme that could be called a "computer immune system," such data may need to be shared with other applications and software components, some of which are described in the following sections.

3.5.1. Subsystems

If fully realized, the virus detection system described herein would consist of three subsystems, each responsible for one aspect of the detection process. One of these components, the learning engine, was partially implemented in order to test the research hypothesis. The following sections describe each of the three subsystems in turn.

3.5.1.1. Learning Engine

This component, some of whose functions are handled by the currently implemented version of DRIVER, is responsible for carrying out the constructive induction process in order to learn more about a particular set of examples. This is how the program learns to distinguish between desirable files and those that may contain viruses or other security threats. Constructive induction revolves around the manipulation of hypotheses, which are generated, ordered, evaluated, and (possibly) incorporated in turn [Gun91]. Generation is the introduction of new hypotheses through the combination of existing attribute values using predefined rules and operators. Ordering is the process of deciding which hypotheses appear to be the most useful and arranging them according to this determination. After this, evaluation takes place, in which all or some of the hypotheses are tested to see if they meet actual system objectives. Finally, those hypotheses that are found to be most useful are incorporated into a list of rules, which may take the form of a data structure. DRIVER is capable of building a binary decision tree for a given group of examples by systematically examining a list of hypotheses and assigning them to the appropriate nodes in the tree.

3.5.1.2. Scanner

The scanner subsystem takes the information gained by the learning engine and uses it to evaluate the target computer. Three possible outcomes may result from the scanning process as applied to any one file. The file will either be a) recognized as an appropriate ("self") piece of code, b) recognized as a known undesirable ("nonself") quantity, or c) fall into an impure leaf in the decision tree, in which even previously seen examples are not classified. In the final case, the file should remain under suspicion until either the user can make a determination as to its validity, or more data can be made available to the learning engine, which will result in positive identification. In its current form, DRIVER does not perform any of the scanning functions. However, it *is* capable of generating decision trees, the efficient data structures that are the basis of the scanning process.

3.5.1.3. Signature Database

Accessed extensively by both the learning engine and the scanner, the signature database is the "data bank" which keeps track of those byte patterns capable of distinguishing between valid and invalid files. Notice that while the learning engine must be capable of writing to this database, the scanner only needs to have read access. The signature database does not directly respond to any user commands; rather, it serves as a repository for the other two subsystems. Although a fully implemented version of the proposed detection system would generate the signature database as a shared, external data entity, DRIVER maintains this list internally for ease of reference.

3.6. Dynamic Structure

One way to describe the behavior of a system is through the set of operations, states, and events that make up its dynamic structure [RBP91]. The following discussion illustrates this structure for both the implemented learning engine component (DRIVER) and a fully functional virus detection system based on the learning engine.

Recall that the stated goals for the virus detection system were to *learn*, *detect*, *adapt*, and *remember*. These goals can be translated almost directly into the system states necessary for accomplishing virus scanning and detection. The first step is for the learning engine component to “learn” which characteristics distinguish acceptable files from those which may not belong in the system, a process required for any system using induction as a learning mechanism. Once a detection baseline has been established, the system passes control to the virus scanning component, which examines files stored locally, whether temporary or not, for any trace of a known virus. From there, one of two conditions—encountering a virus or encountering a file that cannot be classified—will trigger a transition to a new state. If a file containing a virus is found, control should be passed to a component responsible for either removing the file, establishing quarantine, or, if applicable (and possible), restoring the file to a "healthy" state. Such a component, although necessary in any complete antivirus solution, is outside the scope of this research. The other transition event is that in which the scanner finds a file that cannot be classified--the previously-identified scanning outcome (c). In this case, the file will need to be identified through another method (such as examination by a human analyst). Since this third-party identification process may take a relatively large amount of time, the scanner component continues its examination of any remaining files in the interim, flagging subsequent "mystery" files in a similar manner. Once these files have been classified by other means, the results will be introduced to the learning engine component, and the system transitions back to the "learning" state in order to generate a new self-description schema. In a locally-contained detection system such as that described here, a collection of nonself examples could be kept in quarantine for use in learning, with new examples added once they have been identified. Note, however, the important assumption regarding learning. Conditions may occur such that the induction software is unable to learn from the introduction of newly classified files. Consider, for example, the implications of adding a human analyst to the detection loop—errors in classification may cause a contradiction between two or more files, thereby rendering the learning engine incapable of classifying files with similar content (the resulting behavior would resemble that which necessitated a third-party analyst in the first place). These undesirable possibilities notwithstanding, the system will transition to a new state following completion of the learning component's adaptation phase, in which the signature database is updated to reflect the information gained as a result of the third-party classification. Following the completion of this update, control returns once more to the scanner component.

In addition to the behavior exhibited by the virus detection system as a whole, there is also a dynamic structure associated with each of the system's three primary components. The remainder of this section will focus on the structure of the learning engine component, a partial implementation of which exists in the form of the DRIVER prototype. This component is responsible for the learning and adaptation functions performed by the overall system.

The first job that must be accomplished by the learning engine component is to load a group of examples. During this phase, features are extracted from the example files using predefined methods. A *feature* is a collection of attributes (bytes) that can be found inside a particular file. The program then compiles an initial list of *hypotheses*--candidate solutions--from this group of extracted file features. No hypothesis manipulation takes place at this time. This list is then used to perform selective induction over the group of examples, during which a binary decision tree is built. If the selective induction process is successful in classifying the examples without any ambiguity, then the decision tree is saved for use as a detector. If full classification cannot be achieved through selective induction, the learning engine moves to the constructive induction phase, in which new hypotheses are built using a predefined set of operators and the initial list of *selected* hypotheses. Constructive induction may also be invoked in order to achieve greater clarity of representation, or to establish a decision tree with improved inductive generalization power. Note that in an application such as DRIVER, which uses only logical operators for construction, ambiguity cannot be removed; however, a more compact representation can be achieved (see Chapter Five for examples). Following construction, the process of building a decision tree is repeated, using the new list of hypotheses. If the new hypothesis vector proves insufficient to classify the examples, the construct-and-evaluate cycle is repeated, this time generating more complex hypotheses. However, this process *cannot* continue uninterrupted until an unambiguous decision tree is built. Since each new round of the constructive induction process requires an exponentially larger number of steps during both the construction and evaluation states, there are practical limits to the number of times the construct-and-evaluate cycle can be completed. In practice, the learning engine would likely be instructed not to proceed with any further construction once the available hypotheses have reached a specified level of complexity. This is appropriate in light of Occam's Razor, which states that simple theories should be shown preference over those that are more complex. Many additional details regarding this process are covered in Chapter Four. Note that the DRIVER prototype requires user interaction in order to perform all of the steps discussed above, including the extraction of features from example files, which is handled by a separate utility.

3.7. Data Flow

Another way in which to understand the behavior of a system is to examine the flow of data [RBP91]. The primary data types shared among the virus detection system's components are *examples* and *decision trees*, although a decision tree is merely a "packaged" list of one or more hypotheses for use in efficiently classifying examples. Most of the data transformation takes place within the learning engine component, which takes groups of examples and forms a list of hypotheses and a decision tree from the features within those examples. Decision trees are stored in the signature database component, and then passed to the scanner for use in determining which files may contain viruses. Since the detection system will likely encounter, at one point or another, examples which cannot be classified (due to ambiguity in the decision tree), the system must be capable of flagging these anomalies and then accepting data regarding their status, once they have been classified by other means. This information can then be processed by the learning engine and, in turn, added to the signature database.

3.8. Testing and Integration

Since the virus detection system described herein is based on a component architecture, a bottom-up approach to verification testing is appropriate. First, each of the three primary components should be tested to ensure they meet the original operating requirements and that there are no errors in processing. Valid output should be produced for any given set of valid inputs. Once all the components have satisfied these conditions on their own, the next objective is to ensure that the interface between each component works as intended. The signature database must accept hypotheses from the learning engine, the scanner and the learning engine must be able to communicate via the signature database, and the learning engine must accept external inputs such as examples and pre-classified files. Data integrity must be maintained throughout. Finally, the detection system can be tested as a whole.

For research purposes, testing of the DRIVER learning engine prototype focused not only on the efficiency and accuracy of the component's processing, but also on determining the extent to which construction induction is useful in classifying certain test groups designed to simulate real-world detection scenarios.

3.9. Summary

This chapter described, in general, the goals, requirements, and overall architecture of a fully functional virus detection system comprised of three primary components. Coverage was given to important design issues such as dynamic structure, data flow, testing, and integration. Whenever possible, these concepts were also discussed as they apply to DRIVER, an implementation prototype of the learning engine component of the detection system. Deviations between the design proposal and the prototype were noted where appropriate. In the next chapter, DRIVER will be discussed in greater detail. Chapter Five presents results satisfying the research objectives outlined earlier in this chapter.

4. System Implementation

4.1. Overview

Whereas Chapter Three concentrated on the "big picture" regarding a proposed virus detection system, this chapter addresses implementation issues, especially in regard to the working learning engine prototype, DRIVER. These issues include functional decomposition, design decisions and the tradeoffs associated with them, and limitations of the prototype in its current form. The results obtained from testing DRIVER in the laboratory are provided in Chapter Five. Conclusions derived from this research are presented in Chapter Six.

4.2. Definitions

For the reader's convenience, this section collects the definitions for a number of terms frequently referred to throughout this chapter (some of which have already been introduced in preceding chapters).

Attribute	A single byte (8 bits) from a file.
Concept	A description of the classification which the induction process attempts to achieve. This research is primarily concerned with the concepts of "self" and "nonself."
Example	A file or portion of a file that has been labeled in advance as representing self or nonself. The induction engine uses examples to construct decision trees.
Extraction Method	The algorithm used to cull features from files. Three such methods are utilized: chunking, sliding window, and every other sliding window. Also may be referred to as a *Selection Rule.*
Feature	An ordered collection of attributes from a file, collected using one of three extraction methods. The induction engine uses features that are 16 bytes in length.
Hypothesis	All or part of a potential solution to the classification problem which the induction engine attempts to solve. A hypothesis created using selection contains one feature, while a hypothesis created using construction contains two or more features related using one or more operators (see below). Hypotheses selected by the induction engine become signatures.
Operator	A mechanism which relates two or more hypotheses according to a predefined rule. DRIVER uses the logical operators AND, OR, and XOR.
Signature	A series of bytes from a file that can be used to distinguish between self and nonself.

4.3. Induction Engine

Ultimately, DRIVER's function within the virus detection system is to develop the means of distinguishing between acceptable ("self") and unacceptable ("nonself") files, and to provide this information to the other system components. The process by which this distinction is arrived at is constructive induction. The constructive induction process has four stages--generation, ordering, evaluation, and incorporation--all of which revolve around a central data object, the *hypothesis*. The implementation of this process is the topic of the following sections.

4.3.1. Hypothesis Generation

Hypothesis generation is a controlled process, which behaves according to a predefined set of operations. The two methods of generating hypotheses are *selection* and *construction*. Hypotheses that have been created by selection represent features that have been extracted directly from one or more example files according to a set of rules, while hypotheses created by construction represent a composite object formed by joining two other hypotheses with an operator. This section covers these concepts in detail.

4.3.1.1. Hypothesis

The *hypothesis* is the central data object in the induction process. In the abstract, a hypothesis represents a candidate problem solution, or a piece of a solution. Within the virus detection system, hypotheses represent file features (byte strings), or groups of features, which may be used to classify a set of examples.

Although DRIVER performs the induction process using features created by extracting byte sequences from files, byte sequences are not the only characteristics of a file that might be useful in detecting undesirable code. Some antivirus implementations attempt to find viral code by examining the *behavior* of a given system's processes. The reasoning behind this approach is that infected files are likely to cause unusual operating system calls, or place the computer in an otherwise unfamiliar state. This form of observation would require a scanner component different from that described in the previous chapter; however, in principle, the induction engine could continue to perform the same function in such a detection system.

4.3.1.2. Examples

Examples are a crucial component in an inductive learning system. They are the means by which a method of classification with respect to a particular problem domain is formed. DRIVER uses example sets that contain a group of file segments labeled as either self or nonself.

Each example set examined by DRIVER is a single file containing feature data from ten different file segments. This file is built by a custom preprocessor application, which examines a target group of example files, uses assigned extraction methods to generate groups of features (see below), builds a list of unique features, and writes a new file containing both the examples and the feature list. The preprocessor also converts each unique feature into an integer, and, when writing to an example set, represents the byte strings in each example using this integer assignment. The practical benefits of this step should include conservation of both space and engine processing time, although no formal actions were taken to confirm this.

4.3.1.3. Feature Extraction

There are two fundamental methods by which the characteristics of a particular file may be obtained. They may either be *selected* using a particular feature extraction method (many such methods exist), or *constructed* by combining additional features to obtain new features (again, there are an arbitrary number of ways by which to accomplish this). In order to perform induction on a group of examples, DRIVER must start with a group of features that have been selected from those examples. Three such extraction methods were used in obtaining research data:

Table 1 -- Feature Extraction Methods

Method	Definition	Resulting Hypotheses
Chunking	Contiguous byte segments of equal size are extracted from the example (no overlap between segments).	N / K
Sliding Window	Overlapping byte segments of equal size are extracted; the "window" slides one byte toward the end of the file for every subsequent feature selection.	N – K + 1
Every Other Byte Sliding Window	Overlapping byte segments of equal size are extracted; however, only even-numbered bytes in a particular sequence are assigned as attributes of the resulting feature. Therefore, in order to obtain features of the same size as those selected using "sliding window," the window must be twice as large.	N – (2K – 1)

In the table above, *N* represents the total number of bytes in a particular example, and *K* represents the feature size. Based on research precedent, a feature size of 16 byes was chosen [KSSW97].

Since this research seeks to verify findings implied by previous work in induction-based virus detection, the same extraction algorithms used previously have also been employed here. Cardinale and O'Donnell indicated that these three selection rules were chosen based largely on computation and storage limitations. In general, the number of available rules is nearly limitless, and no additional criteria were used in an attempt to find the most effective and/or efficient rules.

The feature selection process described above is not an integrated function of the DRIVER learning engine prototype. Instead, three preprocessing applications were built, one for each extraction method—chunking, sliding window, and every other byte sliding window. Each preprocessing application reads a specified group of examples, selects a series of features based on its respective algorithm, and, using a format understood by DRIVER, writes a new file containing both the original examples and a list of unique features discovered. If desired to facilitate future research, integration of the preprocessors with DRIVER (or another induction engine) could be accomplished with a relatively small amount of effort.

4.3.1.4. Construction

The other method by which features may be obtained from an example is hypothesis construction. Construction is the process of combining hypotheses with constructive operators. This technique is valid for both hypotheses that have been selected directly from an example, as described in the preceding section, *and* hypotheses that have already been generated using construction. In the latter case, the result of construction is simply a more complex hypothesis, containing a greater number of both features and operators.

Just as extraction methods must be chosen in order to perform selection, constructive operators must be utilized in order to perform construction. Operators establish a relationship between existing hypotheses, known as *operands* in this context. In general, these relationships are limited only to those that are testable; that is, the induction process demands that there exist some way of establishing whether a proposed relationship is actually a characteristic of a given example (or not).

The operators used by DRIVER to construct new hypotheses are known as *logical* operators. A Boolean value is the result of an evaluation of such a hypothesis. The three operators, XOR, AND, and OR, are defined below.

Table 2 -- Operator Definitions

Operator	Evaluates To...
XOR	TRUE if and only if one of the operands in the hypothesis evaluates to true and the other evaluates to false
AND	TRUE if both of the operands in the hypothesis evaluate to true; otherwise FALSE
OR	TRUE if at least one of the operands in the hypothesis evaluates to true; otherwise FALSE

The number of hypotheses constructed in one round, from a given set of features and/or hypotheses, can be determined by calculating C (N,2)—"N choose 2"—where *N* is the size of the original set. This assumes that no isomorphs (functionally identical hypotheses) are created as a result of the construction process.

As implemented in DRIVER, several forms of bias exist in the application of operators to existing hypotheses. To begin with, construction is limited to *homogeneous* sets of hypotheses. In this case, that means that no constructed hypothesis is ever composed of features extracted using more than one selection rule. For example, no hypothesis combines a feature selected using chunking with one selected using a sliding window. Not only does this restriction help minimize the total number of constructed hypotheses, it also simplifies the construction process and reduces the complexity associated with the file scanning component of the virus detection system. Another form of bias is represented by the limited choice of operators. The three logical operators utilized by DRIVER (XOR, AND, and OR) were chosen for their ability to capture information from binary files, precedent in virus research, and the relatively high degree of success observed in previous testing conducted by AFIT researchers. In particular, this investigation is concerned with the classification power of these operators when applied to features extracted from various commercial software applications, utilities, and viruses. Another type of bias introduced by DRIVER lies in the procedure by which existing hypotheses and operators are combined. There are a number of possible algorithms by which this combination can be performed, but DRIVER allows only two. The first technique is to apply a single operator (any of the three available) across the entire set of available hypotheses, and then evaluate the results. The second, less conservative technique is to apply *all* of the available operators to the original set of hypotheses before the evaluation step. Note that since the latter technique results in 3(C(N,2)) hypotheses, the amount of time required to perform a complete evaluation also increases by a factor of approximately three.

The actual process used by DRIVER to construct new hypotheses is relatively simple. Using one or more operators as directed by the user, DRIVER starts with a list of all the existing hypotheses. The first hypothesis in the list is then combined with every other hypothesis in the list using one of the designated operators. Each constructed hypothesis is itself a "mini-list" of three components. The first component represents the newly-assigned operator, and the other two components can be thought of as subtrees. The content of a subtree is a hypothesis of any degree of complexity. Following the first round of combinations, the first hypothesis is dropped from the list, and the resulting list of N – 1 components is handed to a new instance of the construction procedure. Construction halts when only one item remains in the list of original hypotheses, and cannot be combined with anything. Then the process is repeated for any remaining operators. Following that, the newly constructed hypotheses are added to a master listing maintained by DRIVER, and considered for inclusion in any subsequent decision trees. Because the construction control structure is functionally equivalent to a nested loop, the construction process is computationally expensive, requiring $O(n^2)$ time for a given list of n original hypotheses.

4.3.2. Hypothesis Evaluation

The hypothesis evaluation step is that which assigns a score to one or more hypotheses based on their classification abilities over a given example domain. The score is based on the results of comparing the hypothesis against an example set to find the extent to which the hypothesis is capable of distinguishing between "self" and "nonself." Based on this score, a hypothesis may be discarded in order to conserve storage space. DRIVER determines the score for a particular hypothesis by calculating a measure of effectiveness known as *entropy*, which captures the amount of information that can be gained by classifying the target examples using that hypothesis [Qui86]. The entropy value is determined by the following equation, in which p is the number of examples classified as members of class P, and n is the number of examples classified as members of class N:

$$\frac{-p\ln(\frac{p}{p+n})-n\ln(\frac{n}{p+n})}{(p+n)}$$

The most desirable entropy score is 0, which is the score assigned to any hypothesis capable of classifying the set of examples so as to completely separate the self examples from the nonself.

Although DRIVER calculates an entropy score for every available hypothesis during the decision tree building phase, it remembers only the lowest scores for each node in the tree. Consequently, an entropy score may be calculated many times for any given hypothesis during the time in which the decision tree is incomplete. If this seems wasteful, keep in mind that the score for any hypothesis may change significantly, depending on which subset of examples has "filtered" down to the point in the decision tree to which the current round of entropy calculations applies.

The method of evaluation used by DRIVER is different from that utilized in previous research. Cardinale and O'Donnell used two measures of effectiveness, power and purity, which also served as the RSM response variables. Power measures the number of examples classified--whether correct or not--against the total number of examples. Purity measures the degree of correctness among examples that were classified [CaO99].

4.3.3. Hypothesis Ordering

The ordering step may be performed either before or after hypothesis evaluation. The primary impetus for ordering hypotheses is to reduce the computational complexity associated with any remaining steps by removing one or more hypotheses from further consideration. Ordering strategies vary, but, as an example, a group of hypotheses may be ordered according to a piece of domain knowledge that has been coded directly into the learning engine. The current DRIVER prototype does not perform hypothesis ordering.

4.3.4. Hypothesis Incorporation

Hypothesis incorporation is that stage in the induction process when one or more hypotheses, either alone or in combination with other hypotheses, are marked as being found most capable of classifying the target example set. In a complete virus detection system, these hypotheses would be converted into a form readable by the other components in the system, reclassified as *signatures*, and added to the signature database. In DRIVER, a hypothesis becomes incorporated when it is added to a decision tree.

4.4. Scanning Using a Decision Tree

Once built, a decision tree is the means by which an induction-based virus detection system would search for suspicious files. Any such fully operational system, using DRIVER as its learning engine, would use virus detectors stored in the form of binary decision trees. Scanning a group of files using a binary decision tree is a straightforward procedure. The scanner component would first need to know what feature extraction method was originally used to create the tree (remember that DRIVER does not mix selection rules when constructing hypotheses; the same is true for trees), so that the distinguishing feature(s) of a particular file will be properly identified. Then, each file would be classified using the criteria specified by the hypotheses located at various nodes within the decision tree. Files containing the feature, or combination of features, described by the first hypothesis would move down one of the branches in the tree, and files not containing the indicated feature(s) would move down the other branch, and so on. The only node in the tree at which a hypothesis *must* appear is node 0, or the "root" of the tree. In most cases, leaves in the decision tree represent points at which a pure subset—a group consisting of either all positive or all negative examples—was isolated during the induction process. The *preferred* result of scanning a previously unseen file is that the new example will fall into one of these leaves in the existing decision tree, and be classified accordingly. The ability to generalize over a larger group of examples is an important product of inductive learning. However, there is always a chance that a new example will not be cleanly classified using an existing decision tree, due to an ambiguity caused by an impure leaf. The virus detection system user may wish to choose a particular posture with regard to such cases. The safest alternative would be to place a quarantine on the file (ban its use by the computer system) until a third-party analysis could be applied to determine the nature of the "unknown" software. Once this analysis is completed, the learning engine component can incorporate this additional knowledge into a new decision tree, and the classification power of the detection system as a whole will grow.

4.5. Summary

This chapter presented some of the implementation and design issues associated with both the DRIVER learning engine prototype and the *proposed* overall virus detection system. The four components of constructive induction—hypothesis generation, evaluation, ordering, and incorporation—were discussed both in the abstract and in how they are accomplished by DRIVER. Limitations and biases inherent in the current implementation were presented, along with ways in which some of the undesirable constraints might be overcome in the future. In Chapter Five, the results of laboratory testing using the DRIVER prototype are presented. Finally, Chapter Six reveals the overall conclusions that can be drawn from this line of research to date, and provides some suggestions as to what types of research may help further advance this particular field of study.

5. Analysis and Results

5.1. Overview

In the previous chapter, the components and algorithms of both a fully operational *proposed* virus detection system and a partially implemented *prototype* learning engine component were discussed in detail. This chapter is primarily concerned with presenting the results obtained by running a series of selected test scenarios using DRIVER, the learning engine prototype. These test scenarios had a dual purpose. The first round of testing was intended to ensure that DRIVER serves its purpose as an induction engine without errors of any kind. The second round of testing involved eleven test cases previously utilized by researchers Cardinale and O'Donnell, in an attempt to verify the intuition they developed for the use of constructive induction in the virus detection domain. Each of the test cases and its focus is explained, although particular emphasis is placed on the final three test cases, in which the example files were derived from software commonly encountered by ordinary users in the "real world." The RSM analysis of these three test cases indicated a tendency for the induction process to select hypotheses based on the XOR and OR operators. DRIVER was able to cross-examine the validity of this analysis by building decision trees based on the given examples, a technique not utilized in experiments conducted by Cardinale and O'Donnell.

5.2. Test Cases

As discussed above, the two primary categories of test cases were those used to test DRIVER for correct operation and output, and those used to validate results obtained through previous research. The latter category can be further divided into laboratory and operational test cases. Regardless of the testing category, particular attention was paid in every test run to the composition of the hypothesis or hypotheses selected by the induction engine for inclusion in the decision tree.

Of those test cases designed to verify DRIVER's operation, the most important was the "weather" data set. This file contained information on 14 different examples, all of which can be classified correctly using both selective induction and constructive induction with logical operators.

The laboratory portion of the test cases consisted of contrived examples designed primarily to test the functionality of the learning engine. Four of these test cases were found to be misrepresented in the literature that documents the research conducted by Cardinale and O'Donnell. The results of these test cases are not emphasized here, but are included for completeness. The operational test cases were put together using byte segments extracted from widely used software applications. Each of the 11 laboratory and operational test cases was made up of 10 file segments, 8 of which were designated as self with the remaining 2 designated as nonself. Because of the overbearing computational complexity introduced by large file sizes and (as a result) numbers of hypotheses, the 10 file segments in each test case were limited to 100 bytes.

5.2.1. Verification Testing

The test case used to perform most of the verification testing on DRIVER was the Weather Example, which is composed of a total of 14 examples, each having four attributes as follows:

Table 3 -- Data for the Weather Example

Example No.	Type	Attrib A	Attrib B	Attrib C	Attrib D
1	nonself	sunny	hot	high	false
2	self	overcast	hot	high	false
3	self	rain	cool	normal	false
4	self	overcast	cool	normal	true
5	self	sunny	cool	normal	false
6	self	sunny	mild	normal	true
7	self	overcast	hot	normal	false
8	nonself	sunny	hot	high	true
9	self	rain	mild	high	false
10	nonself	rain	cool	normal	true
11	nonself	sunny	mild	high	false
12	self	rain	mild	normal	false
13	self	overcast	mild	high	true
14	nonself	rain	mild	high	true

The weather examples can be successfully classified using only selective induction. The resulting decision tree has 6 nodes containing hypotheses and 7 leaves. Since the weather examples can also be classified using constructive induction, the next part of the test was to use the three available operators to generate a batch of all possible hypotheses composed of one operator and two single-feature hypotheses. This process results in 135 new hypotheses (145 altogether). Of those, two--(sunny AND high) and (rain AND true)--are sufficient to construct a new tree with the same classification power as the aforementioned tree containing 6 hypotheses. Thus, a more compact representation of the weather concept has been achieved. However, there is a tradeoff in the amount of time required to establish the tree composed of constructed hypotheses, which is greater than that required to establish a tree composed only of selected hypotheses. Since DRIVER can construct hypotheses of greater complexity than those containing only one operator, a final test was devised to determine whether more sophisticated hypotheses would be of any use in classifying the weather data. The answer to this question turned out to be "yes." A single hypothesis--((sunny AND high) XOR (rain AND true))--is capable, in this case, of successfully separating all self from all nonself examples. Hence, the result of this final test was a decision tree containing only a single node. Unfortunately, the same tradeoff, noted above, between the amount of space required to store the decision tree and the amount of time required to establish the tree was again evident. In fact, the amount of time spent calculating the entropy for each of the 31,330 available hypotheses was several hundred times greater than the amount of time required to establish the tree made up of hypotheses containing a single operator. This test scenario illustrates the need for effective process optimization techniques. Although relatively complex hypotheses with powerful classification properties are desirable (in fact, they may be *required* in other cases), they are much less useful if they cannot be obtained in a reasonable amount of time. Given *a priori* knowledge of this example domain, the time required for DRIVER to identify the most useful classifier may have been reduced significantly.

5.2.2. Laboratory and Operational Test Cases

The eight sets of laboratory test cases were formulated to measure the effectiveness of an induction engine over a wide range of possible conditions, in classification scenarios ranging from obvious to impossible. The following table, copied from the previous work done by Cardinale and O'Donnell, explains the composition of each of these cases:

Table 4 -- Laboratory Test Case Definitions

Number	Structure	Purpose
1	*Self* - All 1s *Nonself* - All 0s	Does detection work?
2	*Self* - All 1s *Nonself* - Random characters	Does the learning engine detect repeated patterns?
3	*Self* - Random characters without y *Nonself* - Random characters with y	Does the learning engine induce a classifier for infrequent patterns?
4	*Self* - Have equal number of 1s and 0s *Nonself* - Random characteristics with unequal number of 1s and 0s	Does the learning engine detect parity of bytes?
5	*Self* - Contains pattern y same distance apart *Nonself* - Contains pattern y varying distance apart	Does the learning engine detect spatially and logically?
6	*Self* - Same as *Nonself*	Does the learning engine's evaluation process work?
7	*Self* and *Nonself* are complements of one another	Can the learning engine induce detectors for absolute position?
8	*Self* - Randomly generated string *Nonself* - Randomly generated string	Does the learning engine detect patterns in random strings?

Examination of the test cases revealed discrepancies between the above descriptions and the actual contents of the original test file segments. In particular, test cases 2, 3, 5, and 8 were misrepresented in the original table shown above. The appropriate descriptions are as follows, in which the test case numbers that required changes are shown in bold print:

Table 5 -- Revised Laboratory Test Case Definitions

Number	Structure
1	*Self* - All 1s *Nonself* - All 0s
2	*Self* - Four file segments composed of all 1s, 4 composed of all 0s *Nonself* - Random characters, both files the same
3	*Self* - Continuous string of '10101010' bytes *Nonself* - Random characters, both files different
4	*Self* - Contain equal number of 1s and 0s *Nonself* - Random characteristics with unequal number of 1s and 0s
5	*Self* - Continuous string of '10101010' bytes *Nonself* - Same as *Self*
6	*Self* - Same as *Nonself*
7	*Self* and *Nonself* are complements of one another
8	*Self* - Randomly generated string, each file segment different *Nonself* - Identical to one of the "self" file segments

Since the test descriptions required modification, the purpose originally behind some of the test cases was invalidated. For example, test case five turned out to be functionally identical to test case six. For that matter, test case eight contained two identical files, one labeled as self and one as nonself, rendering this scenario similar to cases five and six (with the exception of some unambiguous branches in the resulting decision tree).

Each of the eight laboratory test sets was tested using five different sets of hypotheses: selected hypotheses only, selected hypotheses and those constructed using XOR, selected hypotheses and those constructed using AND, selected hypotheses and those constructed using OR, and selected hypotheses combined with those constructed using all available operators. No hypotheses containing more than one operator were constructed. In addition, each of the five sets of hypotheses was generated three times, using each of the three feature extraction methods (chunking, sliding window, and every other byte sliding window). Therefore, a total of 15 tests were run against each group of test examples. The results of this testing were as follows:

Table 6 -- Laboratory Test Case Results

Test Case Number	Results
1	All test groups were classified using a single selected feature (byte '11111111' in self file segments)
2	All test groups were classified using a single selected feature (first byte of the nonself file segments)
3	All test groups were classified using a single selected feature (byte '10101010' in self file segments)
4	A single constructed hypothesis using XOR or OR was selected if available; otherwise, the examples were classified using two selected hypotheses
5	A single selected feature (byte '10101010') was placed at the root of the decision tree; since only one feature was available, the induction engine quit with unclassified examples still remaining
6	The induction engine quit after attempting to classify the identical examples using a very large decision tree
7	All test groups were classified using a single selected feature (first byte of the nonself file segments)
8	Since identical files carrying opposing labels ("self" and "nonself") existed, the induction engine quit after attempting to classify the identical examples using a very large decision tree

The laboratory test cases did not represent an exhaustive exploration of DRIVER's behavior under all possible inputs, but they did provide some data. Test cases five, six, and eight showed that under no circumstances could a group of examples be classified when identical examples labeled as both self and nonself existed within the group--an expected result. In every other case, classification was successful, but the results were not always obtained in accordance with the stated purpose of the individual tests. For example, case seven was intended to test one of the spatial operators used in previous research; however, DRIVER was still successful in classifying the file segments because it recognized the complementary bytes as a distinguishing feature. Case four was a test of the engine's ability to detect byte parity, but because DRIVER simply selected two features present only within the nonself examples, no conclusions could be directly drawn from this test case regarding the ability to detect parity. In some of the other cases, the originally stated goal for the test of the learning engine's behavior was incompatible with the actual contents of the test group, a result of the discrepancies between description and content noted earlier. Group three was a case in point--it was intended as a tool to determine whether the learning engine could find a classifier for infrequent patterns contained within the file segments, but since the "self" examples contained continuously repeating patterns (which DRIVER found), the original question regarding pattern finding could not be answered using this test group.

The operational test cases were created in order to test the learning engine's ability to distinguish between byte segments extracted from actual application programs (and other types of common software). Note, however, that, just as with the laboratory test cases, each of the examples within an operational test case is only 100 bytes in length. In order to obtain the most realistic possible results using such relatively small chunks of code, the following extraction method was specified: start at the beginning of the application file, jump 2,000 bytes toward the end of the file, and then extract the first 100 bytes immediately following the file pointer. The primary goal of this method was to avoid the library references often found in the headers of executable file types, which, if extracted as representative byte patterns, would tend to make groups of examples appear similar to the learning engine. Once again using the descriptions developed by Cardinale and O'Donnell, who created the original data, here is a listing of the operational test cases:

Table 7 -- Operational Test Case Definitions

Number	Structure	Purpose
9	*Self* - Randomly chosen programs *Nonself* - Randomly chosen programs	Does the learning engine detect patterns in programs?
10	*Self* - Programs copyrighted by companies other than Microsoft *Nonself* - Programs copyrighted by Microsoft	Does the learning engine detect patterns in programs from different companies?
11	*Self* - Randomly chosen programs *Nonself* - File infector viruses	Does the learning engine detect viral patterns?

This is the group of examples that, when analyzed using Response Surface Methodology, appeared to be most easily classified using hypotheses containing the XOR and OR operators [CaO99]. This analysis does not necessarily imply that constructive induction is *required* in order to classify the examples—only that hypotheses containing these operators seem to provide a more compact representation of the concept. In fact, since an equivalent decision tree exists for any hypothesis constructed using one of the available logical operators, solutions using only selective induction were known to exist prior to testing. Therefore, DRIVER was directed to use both selective and constructive induction in its analysis of these "real world" test cases. The test series conducted was the same as that described for test cases 1 through 8 above. The goal was to identify any trend in the selection of hypotheses constructed using a particular operator, and also to evaluate the tradeoffs between selective and constructive induction, as described above. Here are the results for these operational test cases:

Table 8 -- Operational Test Case Results

Test Case Number	Results
9	A single constructed hypothesis using XOR or OR was selected if available; XOR was favored if hypotheses containing both operators were available; otherwise, the examples were classified using two selected hypotheses
10	A single constructed hypothesis using XOR or OR was selected if available; XOR was favored if hypotheses containing both operators were available; otherwise, the examples were classified using two selected hypotheses
11	For test groups created using chunking or sliding window as the selection rule, the examples were classified using a single selected feature common to the two viral file segments; for test groups created using every other byte sliding window as the selection rule, hypotheses constructed using XOR were favored, if available (otherwise, the examples could be classified using a single OR hypothesis or two selected features)

A cursory examination of the table above may give the appearance that XOR was the most effective operator across the large majority of the 45 individual test scenarios run using the operational test cases. However, there are several issues regarding both the data and the test results that serve to invalidate this conclusion. First, consider that in those test cases where hypotheses containing XOR were found useful, hypotheses containing the OR operator were also selected, when available. Further, note that for those same test cases, a single hypothesis was sufficient to classify all of the example file segments regardless of whether it was built using OR or XOR. This indicates that the hypotheses containing the OR operator are equivalent in classification power to those containing the XOR operator for these example groups. Why, then, did DRIVER select the hypotheses containing XOR in the scenario in which all possible (single-operator) hypotheses were available? The answer lies in the algorithm used to select from among the available hypotheses. In any batch of hypotheses containing more than one object with equivalent entropy values, DRIVER will retain the first hypothesis found; assuming, of course, that no other available hypotheses exhibit more desirable classification traits. This behavior was confirmed in the laboratory by altering the order in which DRIVER constructs new hypotheses, so that those containing OR were built before those containing XOR. The result was as predicted: a hypothesis containing OR was chosen in place of the previous XOR hypothesis. In the prior research, the use of RSM led to the supposition that XOR and OR were the best candidates for a constructive induction process which attempts to curb computational resource requirements by first generating hypotheses containing the most useful possible operators (of those evaluated up to this point, to be precise). Since those same operators were shown to appear in hypotheses selected by DRIVER, such a proposition, though certainly not demonstrated convincingly, is *not fully inconsistent* with the results obtained here.

However, an examination of the operational test cases themselves suggests that the properties exhibited by such data shed doubt on the usefulness of a constructive induction-based learning engine prototype such as DRIVER. Consider the following data concerning unique features extracted from the test file segments that make up the operational test cases:

Table 9 -- Unique Features in Operational Test Cases

Test Case Number	*Extraction Method*	*Number of Unique Features Discovered (Among 10 File Segments)*
9	Chunking	60
9	Sliding Window	850
9	E/O Byte Sliding Window	690
10	Chunking	60
10	Sliding Window	850
10	E/O Byte Sliding Window	690
11	Chunking	59
11	Sliding Window	821
11	E/O Byte Sliding Window	690

Since the number of features extracted from each 100-byte example file segment using chunking is 6, and the number of file segments per test case is 10, the maximum number of features that can be discovered using chunking--in this case--is 60. This "maximum feature scenario" occurred as a result of the chunking extraction process as applied to both test groups 9 and 10. Furthermore, these same test groups were found to have the maximum possible number of unique features when examined using both of the other extraction methods. In other words, every feature of every file segment in both of the first two operational test cases is different from every other feature of every other file (including the same file) in their respective test groups. This test configuration is inappropriate for validating the usefulness of a constructive induction-based learning mechanism such as DRIVER, since nonself examples can be easily classified using any of the hypotheses selected from among the features found in the nonself examples. The problem can be handled readily by a less powerful learning process such as selective induction. In the case of test groups 9 and 10, the example file segments can be classified using two hypotheses--one of the features selected from each of the two file segments that were labeled as nonself. In general, any group of examples in which all features are unique can be classified using selective induction according to the following hypothesis formula:

$$(f1 \text{ OR } f2 \text{ OR } f3 \ldots \text{ OR } fn)$$

where n is the number of nonself examples and $f1 \ldots fn$ represent features selected from each of the nonself examples. Although the formula above has the appearance of a constructed hypothesis, it is logically equivalent to arranging the set of n selected hypotheses in a decision tree, which is a product of the selective induction process. The following figure depicts this equivalence.

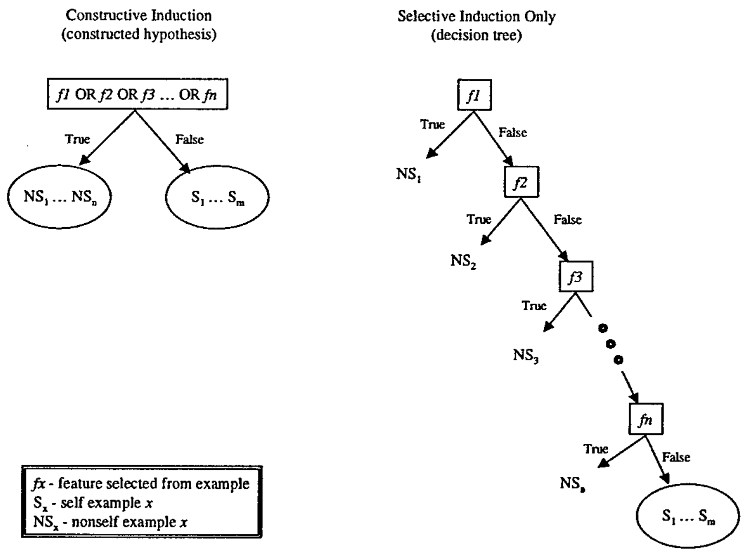

Figure 1 -- Equivalent Representations of a Concept

At first glance, operational test case 11 appears to represent a more functional measure of the learning engine's capabilities, since the unique feature count indicates that there is some feature overlap (recurring features) between the file segments when chunking and sliding window are used. Unfortunately, the amount of overlap is minimal, leaving a significant number of distinct features among the group of file segments. For this reason, test case 11 also does not represent the desired scenario in which the chosen examples cannot be classified using selective induction alone (as shown in the results table).

In addition to the problems with the operational test cases detailed above, attention should be directed toward limitations in scope. Since each of the example file segments in the three test groups was limited to 100 bytes, these segments represent only a very small fraction of the features present in the actual executable files from which they were taken. For example, the Microsoft Word 97 Service Release 2 executable is 5,337,088 bytes in size; therefore, the file segment taken from Word represents only 0.00187% of the "parent" file. One of the goals for future testing of DRIVER, or similar software, would be to increase the size of the individual example files to more realistic proportions. However, a strong possibility exists that the use of larger examples would *not* increase the amount of feature overlap and result in a more desirable testing scenario. The reason for this is the number of unique features made possible by the chunk size used by each of the feature extraction methods. Since a 16-byte chunk contains 128 bits, there are 2^{128} possible chunks, a subset of which will be found in a given example. In other words, unique features will likely be found in even the largest of test examples. The decision tree for a large operational test scenario would therefore contain either a large number of selected features or a smaller number of constructed features (found only at great computational expense). One possible solution to this problem would be to decrease the chunk size used by the feature extraction methods.

Given the information obtained so far regarding the operational test cases, a question arises as to why the RSM analysis performed by Cardinale and O'Donnell indicated a preference for the OR and XOR operators, or even constructive induction in general, since selective induction is suitable in every test case examined. One possible answer is that the learning engine developed in conjunction with the previous research was unable to construct decision trees. In many cases, DRIVER was able to specify a small decision tree (2 nodes and 3 leaves) capable of classifying all of the examples in a particular test group. However, without the ability to build decision trees, a learning engine using algorithms similar to those at work within DRIVER would be unable to discern the value of any one selected hypothesis as used in tandem with another hypothesis. Such a program would be unable to evaluate hypotheses except as standalone classification tools. Hence, constructed hypotheses would be shown preference when examined with respect to their ability to classify the operational test cases used here, since a standalone constructed hypothesis was often sufficient to classify all available examples. However, keep in mind that for the operational test cases, DRIVER found small decision trees that were equivalent to the available constructed hypotheses--no functional difference was found between those hypotheses containing OR and XOR, and the OR operator can be represented using a decision tree. If the RSM analysis had considered decision trees as a guidance mechanism, in addition to individual hypotheses, the indicated preference for constructed hypotheses using XOR and OR may have been reduced, or possibly eliminated altogether.

5.2.3. Future Test Cases

Given the limited utility of the test groups discussed thus far, a need exists to define scenarios better suited for inclusion in subsequent research of this type. This section introduces several hypothetical test cases capable of testing a constructive induction-based learning engine such as DRIVER in a more rigorous fashion.

Excluded Features in Self Examples. Consider a group of test files in which four features exist as follows:

Table 10 -- Proposed Test Case *A*

Example No.	Type	Attrib A	Attrib B	Attrib C	Attrib D
1	self	f1	f4	f4	f4
2	self	f4	f4	f2	f4
3	self	f4	f3	f4	f3
4	self	f2	f4	f4	f2
5	self	f1	f1	f1	f4
6	nonself	f3	f2	f4	f4
7	nonself	f4	f1	f3	f4
8	nonself	f1	f4	f2	f4

Notice that no feature is unique to either the self or the nonself examples. This property prevents selective induction from simply culling distinct features from either the self or nonself examples, which is what happened during many of the experiments using the laboratory and operational test cases. Among the examples shown above, no self example contains features *f1*, *f2*, and *f3* in any combination with one another. In other words, these features are exclusive among the self examples, which also means the entire set can be classified by the hypothesis (*f1* XOR (*f2* XOR *f3*)). Selective induction can be used to build a decision tree equivalent to this hypothesis, but the tree has five nodes and six leaves. Alternatively, given the availability of constructed hypotheses containing one operator, a smaller tree can be built as follows:

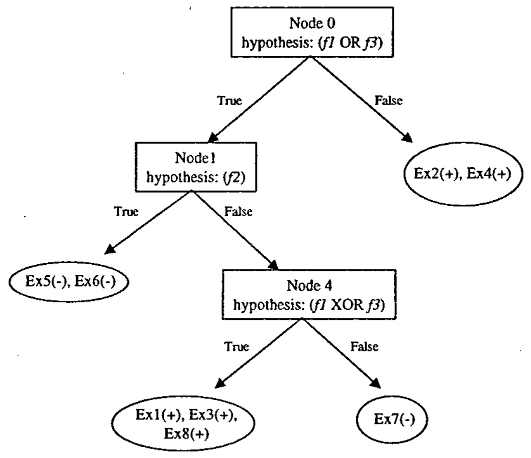

Figure 2 -- Decision Tree for Test Case *A* Using Single-Operator Hypotheses

This test case is similar to the weather example (see Section 5.2.1.) in that a more compact representation can be achieved as more sophisticated hypotheses are made available. Although this scenario used only a small number of examples with a small number of attributes, larger test cases with similar properties could be easily constructed. In principle, this example demonstrates the desired testing situation in which only a few (possibly zero) features are unique to only the self or nonself examples.

Scenarios Testing Additional Operators. If future constructive induction implementations use a different set (or broader range) of operators, then testing strategies appropriate to that program should be devised. The researcher may also desire to create examples testing whether existing operators are capable of "masquerading" as other operators with respect to classification power. For example, the intent originally behind laboratory test case 4 was to test the learning engine's ability to detect byte parity. Although the group of example file segments designated for this test case was, to some extent, inappropriately structured, and no parity-related operator was used in hypothesis construction, the idea of testing for parity recognition is a potentially useful one. In its current implementation, DRIVER can only identify unique features or combinations of features on the byte level, so a complete training set for byte parity would likely contain 2^8 (256) examples and require a decision tree with the same number of leaves. If this turned out to be the case (*and* parity was already known to be a useful classification attribute), then the addition of an operator for even or odd byte parity might be appropriate, since this characteristic can be determined more efficiently than by making eight comparisons (the height of the decision tree in this case) per byte feature. Additionally, consider the previous test scenario. Those examples, which were classified using the operators currently available to DRIVER, could also be classified using an operator which counts the number of unique features in an example, since there were two unique features in every self example and three unique features in every nonself example.

Additional operators may be particularly useful in test scenarios where selective induction is of no use whatsoever. Consider this group of examples:

Table 11 -- Proposed Test Case *B*

Example No.	Type	Attrib A	Attrib B	Attrib C	Attrib D
1	self	f2	f3	f4	f1
2	self	f3	f4	f2	f2
3	self	f1	f2	f3	f4
4	self	f1	f3	f4	f1
5	nonself	f4	f2	f3	f1
6	nonself	f2	f4	f1	f3
7	nonself	f4	f1	f2	f3

So many features are shared among the examples in this test case that they cannot be classified using selective induction. However, close inspection of the self and nonself examples reveals that properties *do* exist which can be exploited by using new operators. Notice that in the self examples, feature *f3* always appears before feature *f4*. This property does not hold in any of the nonself examples. Therefore, a *before* operator could be applied in order to successfully classify the examples in this group. Notice also that only in the self examples do features *f3* and *f4* appear next to one another--they are always separated by at least one other feature in the nonself examples. A *distance* operator could be used to exploit this property. However, one should note that although the appropriate choice of operators can be easily illustrated using an example of this type, such relatively obvious solutions can be very difficult to determine given the large, diverse files likely to be encountered in full-scale test situations. Intimate knowledge of the problem domain is often required in order to make the most effective decisions regarding operator selection and application.

5.3. Performance

Although the measurement of performance was not a primary goal of this research, some remarks as to the performance characteristics displayed by DRIVER are appropriate, since they have some impact on the perceived practicality of the virus detection techniques under investigation. As currently implemented, DRIVER requires extensive system memory resources. The process primarily responsible for these memory requirements is hypothesis construction and storage. Not surprisingly, the test cases in which hypotheses containing all of the available operators are constructed require the most memory. The most intensive of these tests--those in which the maximum possible number of features are extracted from the set of file segments--result in approximately 10^6 hypotheses and require more than 100 MB of storage; a surprising consequence, since DRIVER represents both operators and features internally as 32-bit integers. This exceeds the amount of available physical memory on many current personal computer systems, which is undesirable since applications that require disk swapping typically incur a severe performance penalty. There are two techniques which may allow future versions of DRIVER, or similar software, to function using a smaller amount of an individual computer's resources. The first is to modify the data structure used to contain the list of hypotheses. The current data structure is the "Vector" object supplied with the Java programming language. The second, more extensible technique would be to modify the learning engine so that the hypothesis generation and evaluation phases were accomplished in smaller chunks. At present, the hypothesis evaluation step is not accomplished until all the hypothesis construction is finished. In order to conserve memory, hypothesis construction could be performed in smaller chunks, resulting in an iterative process in which construction and evaluation alternate until the entire batch of hypotheses has been examined (and the best performers retained).

5.4. Summary

In this chapter, the results of the test scenarios using the DRIVER learning engine prototype were presented and evaluated. The original goals of the testing process were to confirm that DRIVER functions properly as a constructive induction engine and to test the research hypothesis that constructive induction is a useful mechanism as applied to virus detection. A set of test cases common to both past and current research efforts was utilized in order to achieve this goal. Although there was some correlation between the previously obtained results and those produced by DRIVER with regard to the selection of hypotheses containing the XOR and OR operators, an evaluation of the data sets indicated deficiencies which limit the conclusions that can be drawn from these results. An important observation with regard to the test data is that for nontrivial test groups, the percentage of unique features among examples is quite high. Reducing the chunk size of the extracted features should result in more feature commonality among examples. The computational complexity inherent in the constructive induction process continues to limit the practicality of the process as implemented here. In Chapter Six, some suggestions for optimizing the process so as to minimize the impact of this issue will be presented, along with other conclusions regarding the data presented in this chapter.

6. Conclusions

6.1. Overview

This research combined aspects of both computer virus detection and inductive machine learning in order to increase the availability of knowledge regarding the extent to which these two concepts can be integrated in a mutually beneficial manner. The overarching goal of this line of research was to demonstrate the advantages and disadvantages of utilizing constructive induction as a means of augmenting currently popular virus detection and eradication techniques. Given the considerable obstacle posed by the computation and storage resource requirements associated with the utilization of constructive induction over example sets of even moderate size, current research should be focused on methods by which the process can be optimized. More efficient applications of the process *should* serve to amplify what is believed to be the primary benefit, which is the ability to generalize over example domains larger than the set of training examples. While the experimental results obtained as a result of this research failed to provide substantive evidence of this benefit, they did consistently confirm the fundamental drawback. A prototype learning engine, presented as one of three components of a proposed *localized* computer virus detection system, was modeled, presented, and implemented in order to investigate some of the real-world ramifications of the overall research goal. The testing performed using this prototype was accomplished in order to obtain evidence either supporting or not supporting the research hypothesis.

6.2. Research Hypothesis

Hypothesis
Constructive induction is an appropriate foundation for the virus detection subsystem of a single-computer antivirus solution

In accordance with this hypothesis, the primary research thrust was to attempt to gather evidence supporting the use of a constructive induction-based learning engine as an integrated component in a virus detection system. No attempt was made to prove or refute the research hypothesis in a formal manner; however, data was obtained which provided some insight into past efforts designed to explore questions regarding the hypothesis. This data was the result of analyzing not only DRIVER's behavior in classifying the examples contained within the eleven test cases, but also the contents of the test cases themselves. Deficiencies in the utility of both the laboratory and operational test cases were found to exist--the unexpectedly limited scope of the laboratory test cases constrained the testing spectrum, and the feature content of the operational test cases was too diverse to be of much use in evaluating the usefulness of DRIVER's ability to produce and evaluate constructed hypotheses. Despite this, the operational test cases were retained for two reasons. First, the chunk size used by the feature extraction methods was likely to result in equally diverse feature content for even much larger example sets. Second, these test cases were useful in providing a direct comparison regarding Cardinale and O'Donnell's conjecture as to the usefulness of response surface methodology. To this end, some correlation was found between the constructed hypotheses chosen by DRIVER for use in classifying the example sets and the operators that the RSM process designated as most useful. However, test runs using DRIVER showed that although hypotheses containing the XOR and OR operators were the most powerful stand-alone classifiers, *constructive induction was an unnecessary computational expense in classifying the operational file segments, given the availability of decision trees.* Therefore, the research hypothesis could neither be proven nor refuted.

In addition, research of this type continues to gather empirical evidence regarding the validity of the research hypothesis. Although DRIVER--a partially-implemented, constructive induction-based learning engine--was able to show instances in which constructed hypotheses displayed classification power beyond that exhibited by selected hypotheses, tradeoffs in the time and space required to arrive at these results had a mitigating effect on the perceived overall usefulness of constructive induction. In practice (and with continuing research), the repeated application of process optimization techniques, combined with variations in other aspects of constructive induction such as operators and feature extraction methods, may yet prove effective in demonstrating constructive induction-based models in which the benefits clearly outweigh the drawbacks.

6.3. Implications

The results of the testing performed using DRIVER indicate that constructive induction, as implemented, appears to be of limited value in the scanning-based virus detection domain. Examples from actual software applications were shown to have very little feature overlap between them, thereby rendering the construction of new hypotheses with logical operators unnecessary. The technique is made even less practical by its computational complexity, the primary drawback to systems based on constructive induction. Domain knowledge can sometimes be utilized to reduce this complexity. Of the many possible pieces of domain knowledge, only one--operator bias--was examined as a possible process optimizer here. The operational test group experiments showed that while not required, hypotheses containing the XOR and OR operators--when available--were chosen by DRIVER as the most capable classifiers in seven of the nine total feature extraction scenarios. While this trend is consistent with indications obtained through previous experiments using RSM, no definitive conclusions have been reached with regard to this optimization technique at this time. In prior research, RSM has been characterized as merely a peripheral to the CI process. However, based on the small amount of evidence gathered using the tests documented herein, RSM continues to show some promise as an *integrated* function in the constructive induction cycle as applied to the domain of computer virus detection.

6.4. Limitations

Computer virus detection and removal is an enormous problem encompassing a diverse array of technical disciplines. Almost any individual research effort devoted to advancing this field is necessarily limited in scope. This research was limited to the investigation of process optimization techniques which *may* be of assistance to a machine learning process, which itself *may* be applicable to the virus detection domain. The models developed in order to conduct this investigation were informal in nature, and can only be verified through informal means.

Beyond those obstacles which befall *any* researcher attempting to tackle one or more of the questions still remaining within the domain of machine learning as applied to virus fighting, each individual research effort must be built around a smaller, but equally important set of constraints. One of the foremost limitations faced by this particular research effort was the limited usefulness of the designated example sets. Although these example sets were needed in order to test the research hypothesis, they were generally inadequate to the tasks of demonstrating the value of DRIVER as a learning component and constructive induction as the basis for a virus detection system. Chapter Five provides some suggestions as to the composition of more robust example sets.

Another important limitation in the study of any constructive induction-based process is the computer hardware on which the testing is performed. Additional operators, features, construction iterations, extraction methods, and test cases all tend to place a considerable burden on the processing and storage subsystems of test machines. For this reason, the number of tests that can be performed over any given length of time tends to be relatively small. Even then, sacrifices may need to be made in the absolute realism of the test scenarios. For example, even though byte sequences from existing software applications were used, at no time was DRIVER involved in testing on "life size" application files. Based on the results of testing using 100-byte example files, the amount of time required to conduct a test based an application such as Microsoft Excel would be truly prohibitive.

6.5. Future Research

- *Incorporation of Response Surface Methodology.* Although the results of this research lend some support to the use of RSM as an integrated constructive induction component, no attempt has yet been made to actually implement the proposed integrated software application. Nothing so far has indicated that doing so does not represent a worthwhile exercise. The performance benefits resulting from the marriage of these two techniques should be measured, and additional testing is required in order to confirm the benefits of RSM over a larger range of test cases.
- *Incorporation of Domain Knowledge.* RSM is not the only optimization technique capable of supplying important pieces of knowledge about a particular problem domain. Other techniques may be equally valuable in providing bits of information leading to a reduction in computational complexity, not the least of which is user experience.

- *Testing of Detection Components Over "Full-Size" Applications.* Using one or more process optimization techniques, it may be possible to conduct virus detection testing using whole application files. However, aggressive complexity reduction measures are required in order to overcome the storage requirements of construction, which increase exponentially with a linear increase in the number of selected hypotheses. Further gains can be accomplished by optimizing the existing DRIVER source code so as to make the currently-implemented constructive induction process more efficient.
- *Exploration of Additional Feature Selection Methods and Operators.* The testing conducted in conjunction with this research utilized only a tiny fraction of the total number of available feature selection methods and operators. There are undoubtedly a number of useful operators that have not yet been explored, and the same holds true for selection methods.

6.6. Summary

Computer viruses remain a tangible threat to systems both within the Department of Defense and throughout the greater international data communications infrastructure on which the DoD increasingly depends. This threat is exacerbated continually, as new viruses are introduced at an alarming rate by the growing collection of connected machines and their operators. Unfortunately, current antivirus solutions are ill-equipped to address these issues in the long term. This thesis documents an investigation into the use of constructive induction, a form of machine learning, as a supplemental antivirus technique theoretically capable of detecting previously unknown viruses through generalized decision-making techniques. A group of examples derived from common software applications, utilities, and viruses was tested in order to evaluate the benefits of adding constructive induction to the process of selecting suitable virus signatures. A prototype virus detection system subcomponent, DRIVER, was developed to conduct the experiments. Due to the feature-rich content of nontrivial example files and DRIVER's ability to assemble decision trees, results showed marginal benefits--compounded with significantly increased computational resource requirements--in the use of constructive induction. Future research, emphasizing a combination of optimization techniques and test cases increasingly approximating "real world" detection scenarios, should eventually establish whether constructive induction represents a genuinely useful and practical alternative to today's antivirus measures.

Appendix A -- Source Code

The source code for DRIVER is not included as part of this document. Those interested in obtaining a copy of the source code should direct their requests to:

Dr. Gregg Gunsch

AFIT/ENG
2950 P Street
WPAFB, OH 45433-7765

gregg.gunsch@afit.af.mil

Bibliography

[AAV99] Dr. Solomon's Virus Central "All About Viruses." Available Online at http://www.drsolomon.com/vircen/vanalyse/va002.cfm

[AOG00] IBM Antivirus Online "Antivirus Online Glossary." Available Online at http://www.av.ibm.com/Education/Glossary/glossary.html

[ABKS94] Aytug, Haldun, Siddhartha Bhattacharyya, Gary J. Koehler and Jane L. Snowdon. "A Review of Machine Learning in Scheduling." *IEEE Transactions on Engineering Management*, 41(2): 165-171, May 1994.

[BlF90] Blanchard, Benjamin S. and Wolter J. Fabrycky. *Systems Engineering and Analysis*. New Jersey: Prentice Hall, 1990.

[BlM99] Bloedorn, Eric and Ryszard S. Michalski. "Data-Driven Constructive Induction in AQ17-PRE." White paper. Center for Artificial Intelligence, George Mason University, 1999.

[BoD87] Box, George E. P. and Norman R. Draper. *Empirical Model-Building and Response Surfaces*. New York: John Wiley & Sons, 1987.

[CaO99] Cardinale, Kelley J. and Hugh M. O'Donnell. *A Constructive Induction Approach to Computer Immunology*. Masters thesis, Air Force Institute of Technology, 1999.

[Che97] Chess, David. "The Future of Viruses on the Internet." In *7th Virus Bulletin International Conference*. Abingdon, England: Virus Bulletin, 1997.

[CLR90] Cormen, Thomas H, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Cambridge: MIT Press, 1990.

[DiM81] Dietterich, Thomas G and Ryszard S. Michalski. "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods." *Artificial Intelligence*, 16:257-294, 1981.

[EAV99] Dr. Solomon's Virus Central "A Guide to Evaluating Antivirus Software." Available Online at http://www.drsolomon.com/vircen/vanalyse/guide_av.cfm

[FHS97] Forrest, Stephanie, Steven A. Hofmeyr, and Anil Somayaji. "Computer Immunology." *Communications of the ACM*, 40(10):88 - 96, October 1997.

[Gun91] Gunsch, Gregg H. *Opportunistic Constructive Inductance: Using Fragments of Domain Knowledge to Guide Construction*. PhD dissertation, University of Illinois at Urbana-Champaign, 1991.

[IUV99] IBM Antivirus Online "Understanding Viruses." Available Online at http://www.av.ibm.com/InsideTheLab/Bookshelf/Understanding

[Kas99] Kaspersky, Eugene. "Viral Analysis Texts: Web version." Metropolitan Network BBS, Inc, 1999.

[KeA94] Kephart, Jeffrey O. and William C. Arnold. "Automatic Extraction of Computer Virus Signatures." In R. Ford, editor, *4th Virus Bulletin International Conference*, pages 179-194. Abingdon, England: Virus Bulletin, 1994.

[Kep94] Kephart, Jeffrey O. "A Biologically Inspired Immune System for Computers." Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pp. 130-139. Cambridge, Mass: MIT Press, 1994.

[KCW83] Kephart, Jeffrey O., David M. Chess, and Steve R. White. *Computers and Epidemiology*. White paper. IBM Thomas J. Watson Research Center, 1983.

[KSSW97] Kephart, Jeffrey O., Gregory B. Sorkin, Morton Swimmer, and Steve R. White. "Blueprint for a Computer Immune System." In Proceedings of the 7th Virus Bulletin International Conference. Abingdon, England: Virus Bulletin, 1997.

[LaS95] Langley, Pat and Herbert A. Simon. "Applications of Machine Learning and Rule Induction." *Communications of the ACM*, 38(11):55-64, November 1995.

[Lud98] Ludwig, Mark. *The Giant Black Book of Computer Viruses*, 2nd Edition. Arizona: American Eagle Publications, 1998.

[Mar97] Martin, James N. *Systems Engineering Guidebook: A Process for Developing Systems and Products*. Boca Raton, FL: Lucent Technologies, 1997.

[Mon91] Montgomery, D.C. *Design and Analysis of Experiments*. Third Edition. New York: John Wiley and Sons, 1991.

[NCS96] NCSA Virus Lab, "Frequently Asked Questions (FAQ) ¼." pp. 7-11, November 29, 1996.

[Pro92] Provost, Foster John. *Policies for the Selection of Bias in Inductive Machine Learning*. PhD thesis, University of Pittsburgh, 1992.

[Qui86] Quinlan, J.R. "Induction of Decision Trees." *Machine Learning*, I:81-106, 1986.

[RBP91] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. New Jersey: Prentice-Hall, 1991.

[Ren90] Rendell, Larry. "Learning Hard Concepts." *Computational Intelligence*. Vol 6, No. 4, 1990.

[TKS96] Tesauro, Gerald, Jeffrey O. Kephart and Gregory B. Sorkin. "Neural Networks for Computer Virus Recognition." *IEEE Expert*, 11(4):5-6, 1996.

[WKC95] White, Steve R., Jeffrey O. Kephart and David M. Chess. "Computer Viruses: A Global Perspective." In Proceedings of the 5th Virus Bulletin International Conference. Abingdon, England: Virus Bulletin, 1995.

[WCC89] White, Steve R., Jimmy Kuo Chengi and David M. Chess. "Coping with Computer Viruses and Related Problems." IBM Thomas J. Watson Research Center, Research Report Number RC 14405, pp. 1-8, January 30, 1989.

[WnM94] Wnek, Janusz and Ryszard S. Michalski. "Hypothesis-Driven Constructive Induction in *AQ17-HCI*." *Machine Learning*, 14:139-168, 1994.

Vitas

Captain Damp was born in Iron River, Michigan, and graduated from the United States Air Force Academy in May of 1995. His first assignment was to the Plans and Programs Division, Communications Directorate, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, where he developed technical solutions satisfying institute communications and computer requirements. Following graduation from AFIT, Captain Damp will be assigned to the Air Force Research Laboratory, Rome, NY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 11-04-2000	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE AN ANALYSIS OF THE EFFECTIVENESS OF A CONSTRUCTIVE INDUCTION-BASED VIRUS DETECTION PROTOTYPE	**5a. CONTRACT NUMBER** EN
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Kevin T. Damp, Captain, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/00J-01
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NM 801 North Randolph Street Room 732 9-65 Arlington VA 22203-1977	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFOSR/NM
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

Advisor: Dr. Gregg H. Gunsch, ENG, DSN: 785-6565 x4281

14. ABSTRACT

Computer viruses remain a tangible threat to the systems on which the Department of Defense increasingly depends. This threat is exacerbated continually, as new viruses are introduced at an alarming rate by the growing collection of connected machines. Unfortunately, current antivirus solutions are ill-equipped to address these issues in the long term. This thesis documents an investigation into the use of constructive induction, a form of machine learning, as a supplemental antivirus technique theoretically capable of detecting previously unknown viruses through generalized decision-making techniques. A group of examples derived from common software applications, utilities, and viruses was tested to evaluate the benefits of utilizing constructive induction in the selection of suitable virus signatures. A prototype virus detection system subcomponent, DRIVER, was developed to conduct the experiments. Due to the feature-rich content of nontrivial example files and DRIVER's ability to assemble decision trees, results showed marginal benefits--compounded with significantly increased computational resource requirements--in the use of constructive induction. Future research, emphasizing a combination of optimization techniques and increasingly realistic test cases, should eventually establish whether constructive induction represents a practical alternative to today's antivirus measures.

15. SUBJECT TERMS

Constructive induction, machine learning, antivirus software, decision tree

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Gregg H. Gunsch
UNCLASS	UNCLASS	UNCLASS	SAR	101	**19b. TELEPHONE NUMBER** *(Include area code)* Comm (937) 255-2024; DSN 785-2024